

# PHarr XCPC Templates

PHarr

2023 年 11 月 26 日



# 目录

<b>第一章 编程技巧和基础算法</b>	<b>7</b>
1.1 读入优化 . . . . .	7
1.2 C++ 标准输入输出 . . . . .	7
1.3 倍增 . . . . .	8
1.4 语法杂项 . . . . .	10
1.4.1 define 与 typedef . . . . .	10
1.4.2 fill 数组填充 . . . . .	10
1.4.3 除法取整 . . . . .	10
1.4.4 map 自定义哈希 . . . . .	11
1.4.5 文件 IO . . . . .	11
1.4.6 乘法爆 long long . . . . .	12
1.5 Lambda 表达式 . . . . .	12
1.6 MT 19937 . . . . .	13
1.7 常用常数 . . . . .	13
1.8 有用的库函数 . . . . .	13
1.9 交互题 . . . . .	14
1.10 Windows 下的对拍器 . . . . .	15
1.11 Linux 下的对拍器 . . . . .	15
1.12 MInt . . . . .	15
<b>第二章 数据结构</b>	<b>19</b>
2.1 并查集 . . . . .	19
2.2 链式前向星 . . . . .	20
2.3 Hash . . . . .	20
2.3.1 Hash 表 . . . . .	20
2.4 栈 . . . . .	21
2.4.1 包含 Min 函数的栈 . . . . .	21
2.4.2 单调栈 . . . . .	21
2.5 ST 表 . . . . .	22
2.6 树状数组 . . . . .	22

2.6.1	单点修改, 区间查询 . . . . .	22
2.6.2	区间修改, 单点查询 . . . . .	23
2.7	分块 . . . . .	24
2.8	ODT . . . . .	26
2.9	线段树 . . . . .	28
2.10	差分 . . . . .	30
2.10.1	二维前缀和、差分 . . . . .	30
2.11	扫描线 . . . . .	31
2.11.1	求面积并 . . . . .	31
2.11.2	二维数点 . . . . .	33
<b>第三章</b>	<b>图论</b>	<b>35</b>
3.1	拓扑排序 . . . . .	35
3.1.1	DFS 算法 . . . . .	35
3.1.2	Kahn 算法 . . . . .	36
3.2	最近公共祖先 (LCA) . . . . .	36
3.2.1	向上标记法 . . . . .	36
3.2.2	倍增 LCA . . . . .	37
3.2.3	RMQ 求 LCA . . . . .	38
3.3	最短路 . . . . .	40
3.3.1	Floyd . . . . .	40
3.3.2	Dijkstra . . . . .	40
3.3.3	Bellman-Ford . . . . .	41
3.3.4	SPFA . . . . .	41
3.4	最小生成树 . . . . .	42
3.4.1	Kruskal . . . . .	42
3.5	树 . . . . .	43
3.5.1	树的深度 . . . . .	43
3.5.2	二叉树还原 . . . . .	43
3.5.3	树的 DFS 序和欧拉序 . . . . .	44
3.5.4	树的重心 . . . . .	45
3.5.5	树上前缀和 . . . . .	45
3.5.6	树上差分 . . . . .	47
3.6	有向图连通性 . . . . .	49
3.6.1	强连通分量 . . . . .	49
3.7	无向图连通性 . . . . .	51
3.7.1	割点 . . . . .	51
3.7.2	桥 . . . . .	51
3.8	图论杂项 . . . . .	52

目录	5
3.8.1 判断简单无向图图	52
3.8.2 2-SAT	53
3.9 二分图	54
3.9.1 二分图	54
3.9.2 二分图最大匹配	55
3.9.3 二分图的最小的点覆盖	57
3.9.4 二分图的最大独立集	57
<b>第四章</b>	<b>59</b>
4.1 数论	59
4.1.1 整除	59
4.1.2 约数	59
4.1.3 GCD 和 LCM	60
4.1.4 质数	61
4.1.5 逆元	62
4.1.6 扩展欧几里得	63
4.2 数学杂项	65
4.2.1 二维向量的叉积	65
4.3 组合数学	66
4.3.1 公式	66
4.3.2 组合数计算	66
4.3.3 公式杂项	67
4.4 线性代数	67
4.4.1 矩阵加速递推	67
4.5 离散数学	69
4.6 计算几何	69
4.6.1 基础模板	69
4.6.2 极角序	78
4.6.3 凸包	78
<b>第五章 字符串</b>	<b>81</b>
5.1 字符串哈希	81
5.1.1 单哈希	81
5.1.2 双哈希	82
5.2 KMP	83
5.3 Tire	84
5.4 最小表示法	85
5.4.1 循环同构	85
5.4.2 最小表示法	85

<b>第六章 动态规划</b>	<b>87</b>
6.1 线性 DP . . . . .	87
6.2 背包 . . . . .	87
6.2.1 01 背包输出方案 . . . . .	87
6.3 树形 DP . . . . .	88
6.3.1 普通树形 DP . . . . .	88
6.3.2 背包类树形 DP . . . . .	89
6.3.3 换根 DP . . . . .	89
6.3.4 最大独立集 . . . . .	91
6.3.5 最小点覆盖 . . . . .	92
6.3.6 最小支配集 . . . . .	92
6.3.7 求任意子树的直径 . . . . .	93
6.4 状态压缩 DP . . . . .	94
6.4.1 TSP 问题 . . . . .	94

# 第一章 编程技巧和基础算法

## 1.1 读入优化

```
1 // 快读
2 int read() {
3     int x = 0, f = 1, ch = getchar();
4     while ((ch < '0' || ch > '9') && ch != '-') ch = getchar();
5     if (ch == '-') f = -1, ch = getchar();
6     while (ch >= '0' && ch <= '9') x = (x << 3) + (x << 1) + ch - '0',
        ch = getchar();
7     return x * f;
8 }
9 // 关闭同步
10 ios::sync_with_stdio(false);
11 cin.tie(nullptr);
```

## 1.2 C++ 标准输入输出

```
1 // 设置输出宽度为 x
2 cout << setw(x) << val;
3
4 // 设置保留小数位数 x , 并四舍五入
5 cout << fixed << setprecision(x) << val;
6
7 //按进制输出
8 cout << bitset<10>(i); // 二进制
9 cout << oct << i; // 八进制
10 cout << dec << i; // 十进制
11 cout << hex << i; // 十六进制
12
```

```

13 // 设置填充符
14 cout << setw(10) << 1234; // 默认是空格
15 cout << setw(10) << setfill('0') << 1234; // 设置填充符
16 //左侧填充
17 cout<<setw(10)<<setfill('0')<<setiosflags(ios::left)<<123;
18 //右侧填充
19 cout<<setw(10)<<setfill('0')<<setiosflags(ios::right)<<123;
20
21 // 单个字符
22 ch = cin.get();
23 cout.put(ch);
24
25 // 指定长度字符串读入
26 cout.get(str,80,'a');// 字符串 字符个数 终止字符
27
28 // 整行读入
29 getlin( cin , s );

```

## 1.3 倍增

天才 ACM

给定一个整数  $M$ ，对于任意一个整数集合  $S$ ，定义“校验值”如下：

从集合  $S$  中取出  $M$  对数 (即  $2 \times M$  个数，不能重复使用集合中的数，如果  $S$  中的整数不够  $M$  对，则取到不能取为止)，使得“每对数的差的平方”之和最大，这个最大值就称为集合  $S$  的“校验值”。

现在给定一个长度为  $N$  的数列  $A$  以及一个整数  $T$ 。我们要把  $A$  分成若干段，使得每一段的“校验值”都不超过  $T$ 。求最少需要分成几段。

1. 初始化  $p = 1$  ,  $r = 1 = 1$
2. 求出  $[1, r+p]$  这一段的校验值，若校验值小于等于  $T$  则  $r += p, p *= 2$ ，否则  $p /= 2$
3. 重复上一步知道  $p$  的值变为 0 此时的  $r$  即为所求

```

1 #include<bits/stdc++.h>
2 #define int long long
3 using namespace std;
4
5 int read() {
6     int x = 0, f = 1, ch = getchar();

```



```
7     while ((ch < '0' || ch > '9') && ch != '-') ch = getchar();
8     if (ch == '-') f = -1, ch = getchar();
9     while (ch >= '0' && ch <= '9') x = (x << 3) + (x << 1) + ch - '0',
        ch = getchar();
10    return x * f;
11 }
12
13 const int N = 5e5+5;
14 int a[N] , b[N];
15 int n , m , t , res;
16 int query( int l , int r ){
17     if( r > n ) return 1e19;
18     for( int i = l ; i <= r ; i ++ ) b[i] = a[i];
19     sort( b + l , b + r + 1);
20     int ans = 0;
21     for( int i = l , j = r , t = 1 ; t <= m && i < j ; t ++ , i ++ , j
        -- )
22         ans += ( b[i] - b[j] ) * ( b[i] - b[j] );
23     return ans;
24 }
25
26 void solve(){
27     n = read() , m = read() , t = read() , res = 0;
28     for( int i = 1 ; i <= n ; i ++ ) a[i] = read();
29     for( int l = 1 , r = 1 , p; l <= n ; l = r + 1 ){
30         p = 1 , r = l;
31         while( p ){
32             if( query( l , r + p ) <= t ) r += p , p *= 2;
33             else p /= 2;
34         }
35         res ++;
36     }
37     cout << res << "\n";
38 }
39
40 int32_t main() {
41     for( int T = read(); T ; T -- ) solve();
42     return 0;
```

43 }

## 1.4 语法杂项

### 1.4.1 define 与 typedef

typedef是用来给类型定义别名，是编译器处理的

#define是字面上进行宏定义用的，是在预处理阶段使用的

```
1 #define STRING char * //宏定义
2 STRING name , sign;//声明
3 char * name , sign;//会被替换成这种的结果,只有 name 是指针
4 // 所以定义类型时候应该回避 #define 而采用typedef
5 typedef char * STRING;
6 char * name , * sign;//会被替换成这种结果
```

### 1.4.2 fill 数组填充

```
1 // 一维数组
2 int a[5];
3 fill( a , a + 5 , 3 );
4
5 // 二维数组
6 int a[5][4];
7 fill( a[0] , a[0] + 5 * 4 , 6 );
8
9 // vector
10 vector<int> a;
11 fill( a.begin() , a.end() , 9 );
```

### 1.4.3 除法取整

```
1 template <typename T, typename U>
2 T ceil(T x, U y) {
3     return (x > 0 ? (x + y - 1) / y : x / y);
4 }
5 template <typename T, typename U>
6 T floor(T x, U y) {
7     return (x > 0 ? x / y : (x - y + 1) / y);
```

```
8 }
```

#### 1.4.4 map 自定义哈希

```
1 // 注意需要以下头文件
2 #include<unordered_map>
3 #include<chrono>
4
5 struct custom_hash {
6     static uint64_t splitmix64(uint64_t x) {
7         // http://xorshift.di.unimi.it/splitmix64.c
8         x += 0x9e3779b97f4a7c15;
9         x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
10        x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
11        return x ^ (x >> 31);
12    }
13
14    size_t operator()(uint64_t x) const {
15        static const uint64_t FIXED_RANDOM = chrono::steady_clock::now
16            ().time_since_epoch().count();
17        return splitmix64(x + FIXED_RANDOM);
18    }
19 };
20 unordered_map<int, int, custom_hash> mp;
```

#### 1.4.5 文件 IO

```
1 # include <fstream>
2
3 // 流对象
4 ifstream fin; // 读入
5 ofstream fout; // 输出
6
7 // 打开
8 ifstream fin("a.in");
9 fin.open("a.in");
10
```

```

11 // 关闭
12 fin.open();
13
14 // 检测 EOF 到达结尾返回非 0 值
15 fin.eof();

```

### 1.4.6 乘法爆 long long

```

1 ll mul(ll x, ll y, ll m) {
2     x %= m, y %= m;
3     ll d = ((long double) x * y / m);
4     d = x * y - d * m;
5     if (d >= m) d -= m;
6     if (d < 0) d += m;
7     return d;
8 }

```

## 1.5 Lambda 表达式

以下内容绝大部分使用与c++14及更新的标准

Lambda 的组成部分是

```
1 [capture] (parameters) mutable -> return-type {statement};
```

首先capture是捕获列表可以从所在代码块中捕获变量。

什么都不写[]就是不进行任何捕获，[=]是值捕获，[&]是引用捕获，值捕获不能修改变量的值，引用捕获可以。特别的，如果值捕获希望在函数内部修改可以使用mutable关键字

同时捕获列表也可以单独针对某一个变量[a]、[&a]分别是值捕获和引用捕获。当然也可以混用[=,&a]对所有变量值捕获，但a除外，a是引用捕获。

然后就是parameters参数列表和statement函数主体，这里与普通的函数没有区别。

-> return-type，函数范围值类型，如果不写可以自动推断，但是如果有多return且返回类型不同就会CE

c++14之后可以用auto来自动的把函数赋值给变量，c++11中则需要自己写

c++17 之后递归可以这样写

```

1 auto dfs = [e](auto &&self, int x) -> void {
2     for (auto y: e[x])
3         self(self, y);
4 };
5 dfs(dfs, 1);

```

## 1.6 MT 19937

mt19937是一个很便捷的随机数生成算法，在 c++11中使用非常便捷

```
1 mt19937 rd(seed) ; // 这样就填入了一个随机数种子
2 rd(); // 这样就会返回一个随机数
3 mt19937_64 rd(); // 相同用法，不过返回是一个 64 位整形
```

## 1.7 常用常数

在<math.h>库中有一些常用的参数

```
1 #if defined _USE_MATH_DEFINES && !defined _MATH_DEFINES_DEFINED
2     #define _MATH_DEFINES_DEFINED
3     #define M_E          2.71828182845904523536    // e
4     #define M_LOG2E      1.44269504088896340736    // log2(e)
5     #define M_LOG10E     0.434294481903251827651    // log10(e)
6     #define M_LN2        0.693147180559945309417    // ln(2)
7     #define M_LN10       2.30258509299404568402    // ln(10)
8     #define M_PI         3.14159265358979323846    // pi
9     #define M_PI_2       1.57079632679489661923    // pi/2
10    #define M_PI_4        0.785398163397448309616    // pi/4
11    #define M_1_PI        0.318309886183790671538    // 1/pi
12    #define M_2_PI        0.636619772367581343076    // 2/pi
13    #define M_2_SQRTPI    1.12837916709551257390    // 2/sqrt(pi)
14    #define M_SQRT2       1.41421356237309504880    // sqrt(2)
15    #define M_SQRT1_2     0.707106781186547524401    // 1/sqrt(2)
16 #endif
```

但是<math.h>并没有默认定义\_USE\_MATH\_DEFINES，所以用之前需要先定义（在万能头下貌似已经被定义过了），在开头加上#define \_USE\_MATH\_DEFINES即可

## 1.8 有用的库函数

1. fabs(x) 用于计算浮点数的绝对值
2. exp(x) 计算  $e^x$
3. log(x) 计算  $\ln(x)$
4. \_\_lg(x) 计算  $\lfloor \log_2(x) \rfloor$

5. `__gcd(x,y)` 计算  $\gcd(x,y)$ , 不过在 C++17 引入了 `gcd(x,y)`, `lcm(x,y)`
6. `ceil(x)` 返回  $\lceil x \rceil$
7. `floor(x)`  $\lfloor x \rfloor$
8. `next_permutation(begin,end)` 将  $[begin, end)$  变为下一个排列, 如果已经是最后一个排列就返回 0
9. `prev_permutation(begin,end)` 将  $[begin, end)$  变为上一个排列, 如果已经是最后一个排列就返回 0
10. `shuffle(begin,end,gen)` 打乱  $[begin, end)$ , `gen` 是一个随机数生成器 (参考 mt19937)
11. `is_sorted(begin,end)` 判断是否升序排序
12. `max(l), min(l)` 对于数组或列表返回最大最小值, 例 `max({x,y,z})`
13. `exp2(x)` 计算  $2^x$
14. `log2(x)` 计算  $\log_2(x)$
15. `hypot(x,y)` 计算  $\sqrt{x^2 + y^2}$
16. `rotate(iterator begin, iterator middle, iterator end)` 作用是把序列中 `begin` 和 `end` 连起来, 然后再从 `middle` 处断开, `middle` 作为新的 `begin`

`__builtin` 家族, 这些内容都是 GNU 私货。如果 `x` 类型是 `long long` 请使用 `__builtin_xxxll(x)`

1. `__builtin_popcount(x)` 返回 `x` 在二进制下 1 的个数。
2. `__builtin_parity(x)` 返回 `x` 在二进制下 1 的个数的奇偶性
3. `__builtin_ffs(x)` 返回 `x` 在二进制下最后一个 1 是从后往前第几位
4. `__builtin_ctz(x)` 返回 `x` 在二进制下后导零的个数
5. `__builtin_clz(x)` 返回 `x` 在二进制下前导零的个数

## 1.9 交互题

这里说的交互题只是 STDIO 交互题

交互题往往是不会限制运行时间的, 一般是通过限制与 oj 的交换次数。对于 IO 交换的题目, 要注意的是在每一次输出后都必须刷新输出缓冲后才可以读入。下面介绍几种语言如何刷新缓冲。

1. C `fflush(stdout)`

2. C++ `fflush(stdout)` 或者 `cout << flush` 或者用 `cout << endl` 输出换行也会自动刷新
3. Java `System.out.flush()`
4. Python `stdout.flush()`

## 1.10 Windows 下的对拍器

```
1 :again
2 data.exe > data.in
3 std.exe < data.in > std.out
4 test.exe < data.in > test.out
5 fc std.out test.out
6 if not errorlevel 1 goto again
```

## 1.11 Linux 下的对拍器

```
1 #!/bin/bash
2 while true; do
3     ./data > data.in
4     ./std <data.in >std.out
5     ./code <data.in >code.out
6     if diff std.out code.out; then
7         printf "AC\n"
8     else
9         printf "Wa\n"
10        exit 0
11    fi
12 done
```

## 1.12 MInt

```
1 template<class T>
2 constexpr T power(T a, i64 b) {
3     T res = 1;
4     for (; b; b /= 2, a *= a)
5         if (b % 2) res *= a;
6     return res;
```

```
7  }
8  template<int P>
9  struct MInt {
10     int x;
11     static int Mod;
12     constexpr MInt() : x(0) {};
13     constexpr MInt(int x) : x(norm(x % getMod())) {};
14     constexpr static void setMod(int Mod_) {
15         Mod = Mod_;
16     }
17     constexpr static int getMod() {
18         if (P > 0) return P;
19         else return Mod;
20     }
21     constexpr int norm(int x) const {
22         if (x < 0) x += getMod();
23         if (x >= getMod()) x -= getMod();
24         return x;
25     }
26     constexpr int val() const {
27         return x;
28     }
29     explicit constexpr operator int() const {
30         // 隐式类型转换把 MInt 转换成 int
31         return x;
32     }
33
34     constexpr MInt operator-() const {
35         MInt res;
36         res.x = norm(getMod() - x);
37         return res;
38     }
39     constexpr MInt inv() const {
40         assert(x != 0);
41         return power(*this, getMod() - 2);
42     }
43     constexpr MInt &operator*=(MInt rhs) &{
44         x = x * rhs.x % getMod();
```



```
45         return *this;
46     }
47     constexpr MInt &operator+=(MInt rhs) &{
48         x = norm(x + rhs.x);
49         return *this;
50     }
51     constexpr MInt &operator-=(MInt rhs) &{
52         x = norm(x - rhs.x);
53         return *this;
54     }
55     constexpr MInt &operator/=(MInt rhs) &{
56         return *this *= rhs.inv();
57     }
58     friend constexpr MInt operator*(MInt lhs, MInt rhs) {
59         MInt res = lhs;
60         res *= rhs;
61         return res;
62     }
63     friend constexpr MInt operator+(MInt lhs, MInt rhs) {
64         MInt res = lhs;
65         res += rhs;
66         return res;
67     }
68     friend constexpr MInt operator-(MInt lhs, MInt rhs) {
69         MInt res = lhs;
70         res -= rhs;
71         return res;
72     }
73     friend constexpr MInt operator/(MInt lhs, MInt rhs) {
74         MInt res = lhs;
75         res /= rhs;
76         return res;
77     }
78
79     friend constexpr std::ostream &operator<<(std::ostream &os, const
80         MInt &a) {
81         return os << a.val();
82     }
```

```
82     friend constexpr std::istream &operator>>(std::istream &is, MInt &
      a) {
83         int v;
84         is >> v;
85         a = MInt(v);
86         return is;
87     }
88
89     friend constexpr bool operator==(MInt lhs, MInt rhs) {
90         return lhs.val() == rhs.val();
91     }
92     friend constexpr bool operator!=(MInt lhs, MInt rhs) {
93         return lhs.val() != rhs.val();
94     }
95 };
96
97 using Z = MInt<998244353>;
```

## 第二章 数据结构

### 2.1 并查集

```
1 // 初始化
2 for( int i = 1 ; i <= n ; i ++ ) fa[i] = i;
3 // 查找
4 int getFa( int x ){
5     if( fa[x] == x ) return x;
6     return fa[x] = getFa( fa[x] );
7 }
8 // 合并
9 void merge( int x , int y ){
10     fa[getFa(x) ] = getFa(y);
11 }
12
13 /*
14  * 下面是一种按秩合并的写法
15  * 简单来说fa[x]<0表示该点为根结点，当x为根节点时 fa[x] = -size[x]
16  */
17 class dsu{
18 private:
19     vector<int> fa;
20 public:
21     dsu( int n = 1 ){
22         fa = vector<int>( n+1 , -1 ) , fa[0] = 0;
23     }
24     int getfa( int x ){
25         if( fa[x] < 0 ) return x;
26         return fa[x] = getfa( fa[x] );
27     }
28     void merge( int x , int y ){
```

```

29         x = getfa(x) , y = getfa(y);
30         if( x == y ) return ;
31         if( fa[x] > fa[y] ) swap( x , y );
32         fa[x] += fa[y] , fa[y] = x;
33     }
34     bool check( int x , int y ){
35         x = getfa(x) , y = getfa(y);
36         return ( x == y );
37     }
38 };

```

## 2.2 链式前向星

链式前向星又名邻接表，其实现在我已经几乎不会再手写链式前向星而是采用vector来代替

```

1 vector<int> e[N]; // 无边权
2 vector< pair<int,int> > e[N]; 有边权
3
4 e[u].push_back(v); // 加边(u,v)
5 e[u].push_back( { v, w } ); //加有权边 (u,v,w)
6 // 无向边 反过来再做一次就好
7
8 for( auto v : e[u] ){ // 遍历
9 }
10 for( auto [ v , w ] : e[u] ) { // 遍历有权边
11 }

```

## 2.3 Hash

### 2.3.1 Hash 表

对数字的 hash

```

1 for( int i = 1 ; i <= n ; i ++ ) b[i] = a[i]; // 复制数组
2 sort( b + 1 , b + 1 + n ) , m = unique( b + 1 , b + 1 + n ) - b; // 排序去重
3 for( int i = 1 ; i <= n ; i ++ ) //hash
4     a[i] = lower_bound( b + 1 , b + 1 + m , a[i] ) - b;

```

除此之外，如果更加复杂的 hash 全部使用unordered\_map容器

## 2.4 栈

### 2.4.1 包含 Min 函数的栈

一个支持  $O(1)$  的 *push()*, *pop()*, *top()*, *getmin()* 的栈  
在维护栈的同时维护一个栈来保存历史上每个时刻都最小值

```
1 struct MinStack{
2     stack<int> a , b;
3     void push( int x ){
4         a.push(x);
5         if( b.size() ) b.push( min( x , b.top() ) );
6         else b.push(x);
7     }
8     void pop(){
9         a.pop() , b.pop();
10    }
11    int top(){
12        return a.top();
13    }
14    int getMin(){
15        return b.top();
16    }
17 };
```

### 2.4.2 单调栈

用来  $O(n)$  的维护出每一个点左侧第一个比他高的点

```
1 int h[N] , l[N]; // h[i] 用来记录每一个点的高度 l[i] 记录每一个左侧第
    一个比 i 高的点的位置
2 stack<int> stk;
3
4 h[0] = INF; //为了便于处理把 0 处理为正无穷
5 stk.push( h[0] );
6 for( int i = 1 ; i <= n ; i ++ ){
7     while( h[i] >= h[ stk.top() ] ) stk.pop();
8     l[i] = stk.top() , stk.push(i);
9 }
```

## 2.5 ST 表

ST 表解决的问题是没有修改且查询次数较多 ( $10^6$ ) 的区间最值查询

$f[i][j]$  表示  $i$  向后  $2^j$  个数的最大值

所以  $f[i][j] = \max( f[i][j-1] , f[ i + ( 1 \ll j-1 ) ][j-1]$

询问首先计算出数最大的  $x$  满足  $2^x \leq r - l + 1$

这样的话  $[l, r] = [l, l + 2^x - 1] \cup [r - 2^x + 1, r]$

```

1 // LOJ10119
2 const int N = 1e6+5 , logN = 20;
3 int a[N] , log_2[N] , f[N][ logN + 5 ];
4 int32_t main() {
5     int n = read() , m = read();
6     for( int i = 1 ; i <= n ; i ++ )
7         a[i] = read();
8     log_2[0] = -1; // 这样初始化可以使得 log_2[1] = 0
9
10    for( int i = 1 ; i <= n ; i ++ ) // O(n) 预处理边界条件 和 log2(i)
11        f[i][0] = a[i] , log_2[i] = log_2[i>>1] + 1;
12
13    for( int j = 1 ; j <= logN ; j ++ )
14        for( int i = 1; i + (1 << j) - 1 <= n ; i ++ )
15            f[i][j] = max( f[i][j-1] , f[ i + ( 1 << j - 1 ) ][j-1] );
16
17    for( int l , r , s ; m ; m -- ){
18        l = read() , r = read() , s = log_2[ r - l + 1 ];
19        printf("%d\n" , max( f[l][s] , f[ r - ( 1 << s ) + 1 ][s] ));
20    }
21    return 0;
22 }
```

## 2.6 树状数组

### 2.6.1 单点修改，区间查询

```

1 struct BinaryIndexedTree{
2     #define lowbit(x) ( x & -x )
3     int n;
4     vector<int> b;
```

```

5
6     BinaryIndexedTree( int n ) : n(n) , b(n+1 , 0){};
7     BinaryIndexedTree( vector<int> &c ){ // 注意数组下标必须从 1 开始
8         n = c.size() , b = c;
9         for( int i = 1 , fa = i + lowbit(i) ; i <= n ; i ++ , fa = i +
            lowbit(i) )
10             if( fa <= n ) b[fa] += b[i];
11     }
12     void add( int i , int y ){
13         for( ; i <= n ; i += lowbit(i) ) b[i] += y;
14         return;
15     }
16
17     int calc( int i ){
18         int sum = 0;
19         for( ; i ; i -= lowbit(i) ) sum += b[i];
20         return sum;
21     }
22 };

```

### 2.6.2 区间修改，单点查询

这里用线段树维护一下差分数组就好

```

1 // op == 1 [l,r] 加上 val
2 // op == 2 查询位置 1 的值
3 int32_t main() {
4     n = read() , m = read();
5     vector<int> t(n+1);
6     for( int i = 1 , x = 0 , lst = 0; i <= n ; i ++ ) x = read() , t[i]
        = x - lst , lst = x ;
7     BinaryIndexedTree B(t);
8     for( int op , l , r , val; m ; m -- ){
9         op = read();
10        if( op == 1 ) l = read() , r = read() , val = read() , B.add(
            l , val ) , B.add( r + 1 , - val );
11        else l = read() , printf("%d\n" , B.calc(l) );
12    }
13    return 0;

```

```
14 }
```

## 2.7 分块

```
1 // https://loj.ac/p/6280
2 #include <bits/stdc++.h>
3 using namespace std;
4 #define int long long
5
6 int read() {...}
7 class decompose {
8 private:
9     struct block {
10         int l, r, sum, tag; // tag是单点修改时的懒惰标记
11         vector<int> val;
12
13         block(int l, int r) : l(l), r(r) {
14             sum = tag = 0;
15             val = vector<int>();
16         }
17     };
18     int len;
19     vector<block> part;
20     vector<int> pos;
21 public:
22     decompose(vector<int> &v) {
23         len = v.size();
24         int t = sqrt(len);
25         pos = vector<int>(len + 1);
26         for (int i = 1; i <= t; i++) // 预处理区间信息
27             part.emplace_back((i - 1) * t + 1, i * t);
28         if (part.back().r < len) // 处理结尾零散部分
29             part.emplace_back(part.back().r + 1, len);
30         for (int i = 1, j = 0; i <= len; i++) {
31             if (i > part[j].r) j++;
32             part[j].val.emplace_back(v[i - 1]);
33             part[j].sum += v[i - 1], pos[i] = j;
34         }
```



```
35     }
36
37     int getSum(int l, int r) {
38         int sum = 0;
39         for (int i = pos[l]; i <= pos[r]; i++) {
40             if (part[i].l >= l && part[i].r <= r) sum += part[i].sum;
41             else
42                 for( auto j = max(l, part[i].l) - part[i].l; j <= min(
43                     r, part[i].r) - part[i].l ; j++ )
44                     sum += part[i].val[j] + part[i].tag;
45         }
46         return sum;
47     }
48
49     void update(int l, int r, int d) {
50         for (int i = pos[l]; i <= pos[r]; i++) {
51             if (part[i].l >= l && part[i].r <= r){
52                 part[i].tag += d;
53                 part[i].sum += part[i].val.size() * d;
54             }
55             else
56                 for (int j = max(l, part[i].l) - part[i].l; j <= min(r
57                     , part[i].r) - part[i].l; j++)
58                     part[i].val[j] += d, part[i].sum += d;
59         }
60     }
61
62 int32_t main() {
63     int n = read();
64     vector<int> a(n);
65     for (auto &i: a) i = read();
66     decompose p(a);
67     for (int opt, l, r, c, sum; n; n--) {
68         opt = read(), l = read(), r = read(), c = read();
69         if (opt == 0)
70             p.update(l, r, c);
```

```

71         else {
72             c++, sum = p.getSum(l, r), sum = (sum % c + c) % c;
73             printf("%lld\n", sum);
74         }
75     }
76     return 0;
77 }

```

## 2.8 ODT

```

1  class ODT{
2  private:
3      struct Node{ // 节点存储结构
4          int l , r;
5          mutable int val;
6          Node( int l , int r = 0 , int val = 0 ) :
7              l(l) , r(r) , val(val){};
8          bool operator < ( Node b ) const {
9              return l < b.l;
10         }
11         int len() const{
12             return r - l + 1;
13         }
14     };
15     int len;
16     set<Node> s;
17 public:
18     // 两种构造函数
19     ODT( const int & n , const vector<int> & a ){
20         len = n;
21         int t = 1;
22         for( int v : a )
23             s.insert( Node( t , t , v ) ) , t ++;
24     }
25     ODT( const int & n , const int a[] ){
26         len = n;
27         for( int i = 1 ; i <= n ; i ++ )
28             s.insert( Node( i , i , a[i] ) );

```

```

29     }
30
31     auto split( int x ){ // 区间分裂
32         if( x > len ) return s.end();
33         auto it = --s.upper_bound( Node( x ) );
34         if( it->l == x ) return it;
35         int l = it->l , r = it->r , v = it->val;
36         s.erase(it);
37         s.insert( Node( l , x-1 , v ) );
38         return s.insert( Node( x , r , v ) ).first;
39     }
40     void assign( int l , int r , int v ){ // 区间赋值 平推操作
41         auto itr = split( r + 1 ) , itl = split( l );
42         s.erase( itl , itr );
43         s.insert( Node( l , r , v ) );
44     }
45     void add( int l , int r , int x ){ // 区间修改
46         auto itr = split( r+1 ) , itl = split( l );
47         for( auto it = itl ; it != itr ; it ++ )
48             it->val +=x;
49     }
50     int rank( int l , int r , int x ){ // 求区间第 x 大
51         auto itr = split(r+1) , itl = split(l);
52         vector< pair<int,int> > v; // first 表示 num , second 表示 cnt
53         for( auto it = itl ; it != itr ; it ++ )
54             v.push_back({ it->val , it->len() } );
55         sort( v.begin() , v.end() );
56         for( auto [ num , cnt ] : v ){
57             if( cnt < x ) x -= cnt;
58             else return num;
59         }
60     }
61     void merge( int l , int r ){ // 区间合并 推平
62         auto itr = split(r+1) , itl = split(l);
63         vector<Node> cur;
64         for( auto it = itl ; it != itr ; it ++ ){
65             if( cur.empty() || it->val != cur.back().val )
66                 cur.push_back( *it );

```

```

67         else cur.back().r = it->r;
68     }
69     s.erase( itl , itr );
70     for( auto it : cur )
71         s.insert( it );
72     return;
73 }
74 int calP( int l , int r , int x , int y ){ // 求区间 x 次方之和
75     auto itr = split(r+1) , itl= split(l);
76     int ans = 0;
77     for( auto it = itl ; it != itr ; it ++ )
78         ans = ( ans + power(it->val,x,y) * it->len()%y ) % y;
79     return ans % y;
80 }
81 };

```

## 2.9 线段树

```

1  #include <bits/stdc++.h>
2  using namespace std;
3
4  #define int long long
5
6  const int N = 5e5+5;
7  int n , m , a[N];
8
9  struct Node{
10     int l , r , value , add;
11     Node * left , * right;
12     Node( int l , int r , int value , int add , Node * left , Node *
        right ) :
13         l(l) , r(r) , value(value) , add(add) , left(left) , right
            (right) {};
14 } * root;
15
16 int read() {...}
17 // 建树
18 Node * build( int l , int r ){

```

```
19     if( l == r ) return new Node( l , r , a[l] , 0 , nullptr , nullptr
        );
20     int mid = ( l + r ) >> 1;
21     Node * left = build( l , mid ) , * right = build( mid + 1 , r );
22     return new Node( l , r , left->value+right->value,0,left , right);
23 }
24 // 标记
25 void mark( int v , Node * cur ){
26     cur -> add += v;
27     cur -> value += v * ( cur->r - cur->l + 1 );
28 }
29 // 标记下传
30 void pushdown( Node * cur ){
31     if( cur -> add == 0 ) return;
32     mark( cur -> add , cur -> left ) , mark( cur -> add , cur -> right
        );
33     cur -> add = 0;
34 }
35 // 修改
36 void modify( int l , int r , int v , Node * cur ){
37     if( l > cur -> r || r < cur -> l ) return;
38     if( l <= cur -> l && r >= cur -> r ){
39         mark( v , cur );
40         return;
41     }
42     pushdown( cur );
43     int mid = ( cur -> l + cur -> r ) >> 1;
44     if( l <= mid ) modify( l , r , v , cur -> left );
45     if( r > mid ) modify( l , r , v , cur -> right );
46     cur -> value = cur ->left->value + cur ->right->value;
47     return ;
48 }
49 // 查询
50 int query( int l , int r , Node * cur ){
51     if( l <= cur->l && r >= cur ->r ) return cur->value;
52     pushdown( cur );
53     int mid = ( cur->l + cur->r ) >> 1 , res = 0;
54     if( l <= mid ) res += query( l , r , cur->left );
```

```

55     if( r > mid ) res += query( l , r , cur->right );
56     return res;
57
58 }
59
60 int32_t main(){
61     n = read() , m = read();
62     for( int i = 1 ; i <= n ; i ++ ) a[i] = read();
63     root = build( 1 , n );
64     for( int i = 1 , opt , l , r ; i <= m ; i ++ ){
65         opt = read() , l = read() , r = read();
66         if( opt == 1 ) modify( l , r , read() , root );
67         else printf("%lld\n" , query( l , r , root ) );
68     }
69     return 0;
70 }

```

## 2.10 差分

### 离散化差分

```

1 // 离散化差分这里我用map实现
2 map<int,int> b;
3
4 void update( int l , int r , int v ){
5     b[l] += v , b[r+1] -= v;
6 }
7
8 // 求和
9 for( auto it = b.begin() ; next(it) != b.end() ; it = next(it) )
10     next(it)->second += it->second;

```

### 2.10.1 二维前缀和、差分

```

1 // 二维前缀和
2 b[i][j] = b[i-1][j]+b[i][j-1]-b[i-1][j-1] + a[i][j];
3
4 // 求 (x1,y1) ~ (x2,y2) 和
5 b[x2][y2] - b[x2][y1-1] - b[x1-1][y2] + b[x1-1][y1-1];

```

## 2.11 扫描线

### 2.11.1 求面积并

给若干个矩形，求面积的和。这里通常会对下标进行哈希

```
1 // luogu P5490
2 #include<bits/stdc++.h>
3
4 using namespace std;
5
6 #define int long long
7 using vi = vector<int>;
8
9 struct Node {
10     int l, r, value, cnt; // cnt 指当前区间被覆盖的次数
11     Node *left, *right;
12
13     Node(int l, int r, int value, int cnt, Node *left, Node *right)
14         : l(l), r(r), value(value), cnt(cnt), left(left), right(
            right) {}
15 } *root;
16
17
18 using seg = array<int, 4>;
19
20 vi raw; // 指原始坐标
21
22 int val(int x) { // x 是原始值，返回哈希值
23     int i = lower_bound(raw.begin(), raw.end(), x) - raw.begin();
24     if (raw[i] != x) return -1;
25     return i;
26 }
27
28
29 Node *build(int l, int r) {
30     if (l == r) return new Node(l, r, 0, 0, nullptr, nullptr);
31     int mid = (l + r) / 2;
32     auto left = build(l, mid), right = build(mid + 1, r);
33     return new Node(l, r, 0, 0, left, right);
```

```
34 }
35
36 void pushUp(Node *rt) {
37     if (rt->cnt > 0) rt->value = raw[rt->r + 1] - raw[rt->l];
38     else if (rt->left == nullptr) rt->value = 0;
39     else rt->value = rt->left->value + rt->right->value;
40     return;
41 }
42
43 void modify(Node *rt, int l, int r, int v) {
44     if (l > rt->r or r < rt->l) return;
45     if (l <= rt->l and rt->r <= r) {
46         rt->cnt += v;
47         pushUp(rt);
48         return;
49     }
50     int mid = (rt->l + rt->r) / 2;
51     if (l <= mid) modify(rt->left, l, r, v);
52     if (mid < r) modify(rt->right, l, r, v);
53     pushUp(rt);
54 }
55
56 int32_t main() {
57     ios::sync_with_stdio(false) , cin.tie(nullptr);
58     int n;
59     cin >> n;
60     vector<seg> p;
61     for (int i = 1, xa, xb, ya, yb; i <= n; i++) {
62         cin >> xa >> ya >> xb >> yb;
63         p.push_back(seg{xa, ya, yb, 1});
64         p.push_back(seg{xb, ya, yb, -1});
65         raw.push_back(ya), raw.push_back(yb);
66     }
67     sort(p.begin(), p.end());
68     sort(raw.begin(), raw.end());
69     raw.resize(unique(raw.begin(), raw.end()) - raw.begin());
70     int tot = raw.size() - 1;
71     root = build(0, tot);
```



```

72     modify(root, val(p[0][1]), val(p[0][2]) - 1, p[0][3]);
73     int res = 0;
74     for (int i = 1; i < p.size(); i++) {
75         res += (p[i][0] - p[i - 1][0]) * root->value;
76         modify(root, val(p[i][1]), val(p[i][2]) - 1, p[i][3]);
77     }
78     cout << res << "\n";
79     return 0;
80 }

```

### 2.11.2 二维数点

单纯的二维数点数点问题，可以只用树状数组就可以维护。

$d(x, y)$  表示从  $(0, 0)$  到  $(x, y)$  中点的数量，因此从左下角  $(a, b)$  到右上角  $(c, d)$  中点的数量就可以表示为  $d(c, d) - d(c, b - 1) - d(a - 1, d) + d(a - 1, b - 1)$ ，这个形式就是普通的二维前缀和。我们把式子稍作变形转换为  $d((c, d) - d(c, b - 1)) - (d(a - 1, d) - d(a - 1, b - 1))$  这样的话就可以用扫描线优化掉一维。

```

1  #include <bits/stdc++.h>
2
3  using namespace std;
4
5  using vi = vector<int>;
6  using pii = pair<int, int>;
7  using seg = array<int, 5>;
8
9  #define lowbit(x) ( x & -x )
10
11 const int N = 1e7 + 5;
12
13 struct BinaryIndexedTree {...}; // 树状数组板子
14
15
16 int32_t main() {
17     ios::sync_with_stdio(false), cin.tie(nullptr);
18     int n, m;
19     cin >> n >> m;
20     vector<pii> a(n);
21     for (auto &[x, y]: a)

```

```
22         cin >> x >> y, x++, y++;
23     vector<seg> p;
24     for (int i = 0, xa, xb, ya, yb; i < m; i++) {
25         cin >> xa >> ya >> xb >> yb;
26         xa++, ya++, xb++, yb++;
27         p.push_back(seg{xb, ya, yb, i, 1});
28         p.push_back(seg{xa - 1, ya, yb, i, -1});
29     }
30
31     sort(p.begin(), p.end());
32     sort(a.begin(), a.end());
33     vi res(m);
34     BinaryIndexedTree bit(N);
35     for (int j = 0; const auto &it: p) {
36         while (j < a.size() and a[j].first <= it[0])
37             bit.update(a[j].second, 1), j++;
38         res[it[3]] += it[4] * bit.calc(it[1], it[2]);
39     }
40
41     for (auto i: res)
42         cout << i << "\n";
43     return 0;
44 }
```

## 第三章 图论

### 3.1 拓扑排序

在一个 DAG 中，将图中的顶点以线性的方式排序，使得对于任何一条  $u$  到  $v$  的有向边， $u$  都可以出现在  $v$  的前面

**拓扑序判环**如果图中已经没有入度为零的点但是依旧有点时，即说明图中存在环。

**拓扑序判链**如果求拓扑序的过程中队列中同时存在两个及以上的元素，说明拓扑序不唯一，不是一条链

**字典序最大、最小的拓扑序**把 Kahn 中队列换成是大根堆、小根堆实现的优先队列就好

#### 3.1.1 DFS 算法

```
1  vector<int> e[N]; // 邻接表
2  vector<int> topo; // 存储拓扑序
3  set<int> notInDeg; // 储存没有入读的点
4  int vis[N];
5
6  bool dfs( int u ){
7      vis[u] = 1;
8      for( auto v : e[u] ){
9          if( vis[v] ) return 0;
10         if( !dfs(v) ) return 0;
11     }
12     topo.push_back(u);
13     return 1;
14 }
15
16 bool topSort(){
17     if( notInDeg.empty() ) return 0 ;
18     for( int u : notInDeg ){
19         if( !dfs(u) ) return 0;
20     }
```

```

21     reverse( topo.begin() , topo.end() );
22     return 1;
23 }

```

### 3.1.2 Kahn 算法

```

1  vector<int> e[N]; // 邻接表
2  vector<int> topo; // 存储拓扑序
3  int inDeg[N]; // 记录点的当前入度
4
5  bool topSort(){
6      queue<int> q;
7      for( int i = 1 ; i <= n ; i ++ )
8          if( inDeg[i] == 0 ) q.push(i);
9      while( q.size() ){
10         int u = q.front() ; q.pop();
11         topo.push_back(u);
12         for( auto v : e[u] ){
13             if( --inDeg[v] == 0 ) q.push(v);
14         }
15     }
16     return topo.size() == n;
17 }

```

## 3.2 最近公共祖先 (LCA)

### 3.2.1 向上标记法

向上标价法最差的复杂度是  $O(N)$  的，适用于求 LCA 次数少的情况，代码非常好写

```

1  \\ luogu P3379
2  const int N = 5e5+5;
3  int n , m , sta , dep[N] , fa[N];
4  vector<int> e[N];
5
6  int read() {...}
7  void dfs( int x ){
8      for( auto v : e[x] ){
9          if( v == fa[x] ) continue;

```

```

10         dep[v] = dep[x] + 1 , fa[v] =x;
11         dfs(v);
12     }
13 }
14
15 int lca( int x , int y ){
16     if( dep[x] < dep[y] ) swap( x , y );
17     while( dep[x] > dep[y] ) x = fa[x];
18     while( x != y ) x = fa[x] , y = fa[y];
19     return x;
20 }
21
22 int32_t main() {
23     n = read() - 1 , m = read() , sta = read();
24     for( int u , v ; n ; n -- )
25         u = read(),v = read() , e[u].push_back(v) , e[v].push_back(u);
26     dep[sta] = 0 , fa[sta] = sta;
27     dfs( sta );
28     for( int x , y ; m ; m -- ){
29         x = read() , y = read();
30         cout << lca(x,y) << endl;
31     }
32     return 0;
33 }

```

### 3.2.2 倍增 LCA

值得注意的是向上倍增的过程中for( int i = t ; i >= 0 ; i --) 不能写错

```

1  const int N = 5e5+5;
2  int n , m , sta , logN , dep[N] , fa[N][20];
3  vector<int> e[N];
4  int read() {...}
5
6  void dfs( int x ){
7      for( auto v : e[x] ){
8          if( dep[v] ) continue;
9          dep[v] = dep[x] + 1 , fa[v][0] = x;
10         for( int i = 1 ; i <= logN ; i ++ )

```

```

11         fa[v][i] = fa[ fa[v][i-1] ][i-1];
12     dfs(v);
13 }
14 }
15 int lca( int x , int y ){
16     if( dep[x] > dep[y] ) swap( x , y );
17     for( int i = logN ; i >= 0 ; i -- )
18         if( dep[ fa[y][i] ] >= dep[x] ) y = fa[y][i];
19     if( x == y ) return x;
20     for( int i = logN ; i >= 0 ; i -- ){
21         if( fa[x][i] != fa[y][i] ) x = fa[x][i] , y = fa[y][i];
22     }
23     return fa[x][0];
24 }
25
26 int32_t main() {
27     n=read()-1 , m=read() , sta=read() , logN =(int)log2(n) + 1;
28     int k = n;
29     for( int u , v ; n ; n -- )
30         u=read() , v=read() , e[u].push_back(v) , e[v].push_back(u);
31     dep[sta] = 1;
32     dfs( sta );
33     for( int x , y ; m ; m -- ){
34         x = read() , y = read();
35         cout << lca(x,y) << endl;
36     }
37     return 0;
38 }

```

### 3.2.3 RMQ 求 LCA

dfs 求出深度和 dfn。位置在  $[dfn_{u+1}, dfn_v]$  之间深度最小的节点的父亲就是 LCA。区间最值用 ST 表维护。

```

1 // luogu P3379
2 #include <bits/stdc++.h>
3 using namespace std;
4
5 int dn;

```

```

6  vector<int> dfn;
7  vector <vector<int>> e, f;
8
9  void dfs(int x, int fa) {
10      f[0][dfn[x] = ++dn] = fa;
11      for (auto y: e[x])
12          if (y != fa) dfs(y, x);
13  }
14  int get(int x, int y) {
15      if (dfn[x] < dfn[y]) return x;
16      return y;
17  }
18  int lca(int u, int v) {
19      if (u == v) return u;
20      u = dfn[u], v = dfn[v];
21      if (u > v) swap(u, v);
22      int d = log2(v - u++);
23      return get(f[d][u], f[d][v - (1 << d) + 1]);
24  }
25  int32_t main() {
26      ios::sync_with_stdio(0), cin.tie(0);
27      int n, m, root, logN;
28      cin >> n >> m >> root;
29      e.resize(n + 1), dfn.resize(n + 1);
30      logN = log2(n);
31      f = vector(logN + 1, vector<int>(n + 1));
32      for (int i = 1, x, y; i < n; i++)
33          cin >> x >> y, e[x].push_back(y), e[y].push_back(x);
34      dfs(root, 0);
35
36      for (int i = 1; i <= logN; i++)
37          for (int j = 1; j + (1 << i) - 1 <= n; j++)
38              f[i][j] = get(f[i - 1][j], f[i - 1][j + (1 << i - 1)]);
39
40      for (int x, y; m; m--) cin >> x >> y, cout << lca(x, y) << "\n";
41      return 0;
42  }

```

## 3.3 最短路

### 3.3.1 Floyd

```

1  int dis[N][N];
2  for( int i = 1 ; i <= n ; i ++ )
3      for( int j = 1 ; j < i ; j ++ )
4          dis[i][j] = dis[j][i] = inf;
5
6  for( int u , v , w ; m ; m -- )
7      u = read() , v = read() , w = read() , dis[u][v] = dis[v][u] = w;
8  for( int k = 1 ; k <= n ; k ++ )
9      for( int i = 1 ; i <= n ; i ++ )
10         for( int j = 1 ; j < i ; j ++ )
11             f[i][j] = f[j][i] = min( f[i][j] , f[i][k] + f[k][j] );

```

### 3.3.2 Dijkstra

复杂度是  $O(m \log(n))$

```

1  void dij(){
2      for( int i = 1 ; i <= n ; i ++ ) dis[i] = inf;
3      dis[sta] = 0;
4      priority_queue< pair<int,int> , vector<pair<int,int>> , greater<
        pair<int,int>> > q;
5      q.push( { 0 , sta } );
6      while( q.size() ){
7          int u = q.top().second ; q.pop();
8          if( vis[u] ) continue;
9          vis[u] = 1;
10         for( auto [v,w] : e[u] )
11             if( dis[v] > dis[u] + w ){
12                 dis[v] = dis[u] + w;
13                 q.push( {dis[v] , v} );
14             }
15     }
16 }

```



### 3.3.3 Bellman-Ford

复杂度是  $O(km)$

```

1 bool bellmanFord(){ // 返回是否有最短路
2     for( int i = 1 ; i <= n ; i ++ ) dis[i] = inf;
3     dis[sta] = 0;
4     bool flag;
5     for( int i = 1 ; i <= n ; i ++ ){
6         flag = 0;
7         for( int u = 1 ; u <= n ; u ++ ){
8             if( dis[u] == inf ) continue; // 如果当前点和起点没有联
              通，就无法进行松弛操作
9             for( auto [v,w] : e[u] ){
10                 if( dis[v] <= dis[u] + w ) continue;
11                 dis[v] = dis[u] + w , flag = 1; // 记录时候进行松弛操作
12             }
13         }
14         if( !flag ) break;
15     }
16     return flag;
17 }
```

### 3.3.4 SPFA

没有准确复杂度，下限是  $O(m \log(n))$ ，上限是  $O(nm)$

```

1 int dis[N] , cnt[N];
2 vector< pair<int,int> > e[N];
3 bitset<N> vis;
4
5 bool spfa(){
6     for( int i = 1 ; i <= n ; i ++ ) dis[i] = inf;
7     queue< int > q;
8     dis[sta] = 0 , vis[sta] = 1 , q.push(sta);
9     for( int u ; q.size() ; ){
10         u = q.front() , q.pop() , vis[u] = 0;
11         for( auto [ v , w ] : e[u] ){
12             if( dis[v] <= dis[u] + w ) continue;
13             dis[v] = dis[u] + w ;
14             cnt[v] = cnt[u] + 1; // 记录最短路经过了几条边
```

```

15         if( cnt[v] >= n ) // 最短路最长是 n-1
16             return false; // 此时说明出现了 负环
17         if( !vis[v] ) vis[v] = 1 , q.push(v);
18     }
19 }
20 return true;
21 }

```

## 3.4 最小生成树

### 3.4.1 Kruskal

Kruskal 总是维护无向图的最小生成森林。

```

1 // Luogu P3366
2 const int N = 5005;
3 int n , m , fa[N] , cnt = 0 , sum;
4
5 int read() {...}
6
7 int getfa( int x ){...}
8
9 void merge( int x , int y ){...} // 并查集合并
10
11 int32_t main(){
12     n = read() , m = read();
13     for( int i = 1 ; i <= n ; i ++ ) fa[i] = i;
14     vector<tuple<int,int,int>> e(m);
15     for( auto & [ w , u , v ] : e )
16         u = read() , v = read() , w = read();
17     sort( e.begin() , e.end() );
18     for( auto [ w , u , v ] : e ){
19         if( getfa(u) == getfa(v) ) continue;
20         merge( u , v ) , sum += w , cnt ++;
21         if( cnt == n - 1 ) break;
22     }
23     if( cnt == n - 1 ) cout << sum << "\n";
24     else cout << "orz\n"; // 图不联通
25     return 0;

```

26 }

最大生成树只需要把边从大到小选择就好

## 3.5 树

### 3.5.1 树的深度

```
1 dep[sta] = 1;
2
3 void dfs( int u ){
4     for( auto v : e[u] ){
5         if( dep[v] ) continue;
6         dep[v] = dep[u] + 1;
7         dfs( v );
8     }
9 }
10
11 dfs(sta);
```

### 3.5.2 二叉树还原

```
1 struct Node {
2     int v;
3     Node *l, *r;
4     Node(int v, Node *l, Node *r) : v(v), l(l), r(r) {};
5 };
6 // 根据中序、后序还原二叉树
7 Node *build(vector<int> mid, vector<int> suf) {
8     int v = suf.back();
9     Node *l = nullptr, *r = nullptr;
10    int t;
11    for (t = 0; t < mid.size(); t++)
12        if (mid[t] == v) break;
13    auto midll = mid.begin();
14    auto midlr = mid.begin() + t;
15    auto midrl = mid.begin() + t + 1;
16    auto midrr = mid.end();
17    auto sufll = suf.begin();
```

```

18     auto suflr = suf.begin() + t;
19     auto sufrr = suf.begin() + t;
20     auto sufrr = suf.end() - 1;
21
22     auto midl = vector<int>(midll, midlr);
23     auto midr = vector<int>(midrl, midrr);
24     auto sufl = vector<int>(sufl1, suflr);
25     auto sufr = vector<int>(sufrr1, sufr);
26
27     if (!midl.empty()) l = build(midl, sufl);
28     if (!midr.empty()) r = build(midr, sufr);
29
30     return new Node(v, l, r);
31 }

```

### 3.5.3 树的 DFS 序和欧拉序

DFS 序中一个点会出现一次，欧拉序会出现两次，两者都是 dfs 时经过的点的顺序，欧拉序中的第二次出现就是点结束搜索是回溯时的顺序

```

1 // 求 DFS 序
2 vector<int> dfsSort;
3 vector<int> e[N];
4 bitset<N> vis;
5
6 void dfs( int x ){
7     dfsSort.push_back(x) , vis[x] = 1;
8     for( auto it : e[x] ){
9         if( vis[it] ) continue;
10        dfs( it );
11    }
12 }
13 // 求欧拉序
14 vector<int> eulerSort;
15 vector<int> e[N];
16 bitset<N> vis;
17
18 void dfs( int x ){
19     eulerSort.push_back(x) , vis[x] = 1;

```

```

20     for( auto it : e[x] ){
21         if( vis[it] ) continue;
22         dfs( it );
23     }
24     eulerSort.push_back(x);
25 }

```

### 3.5.4 树的重心

对于`size[i]`表示每个点子树大小,`max_part(x)`表示删去`x`后最大的子树的大小,`max_part`取到最小值的点`p`就是树的重心

```

1 void dfs( int u ){
2     vis[u] = size[u] = 1;
3     for( auto v : e[u] ){
4         if( vis[v] ) continue;
5         dfs(v);
6         size[u] += size[v];
7         max_part = max( max_part , size[v] );
8     }
9     max_part = max( max_part , n - size[u] );
10    if( max_part > ans )
11        ans = max_part , pos = u;
12 }

```

### 3.5.5 树上前缀和

设`sum[i]`表示节点`i`到根节点的权值总和。

如果是点权,`x,y`路径上的和为`sum[x]+sum[y]-sum[lca]-sum[fa[lca]]`

```

1 // Loj 2491
2 // 一颗树根节点是 1 , 点权就是深度的 k 次方
3 // m次询问,每次问(u,v)路径上点权之和
4 // k 每次都不同但是取值范围只有[1,50]
5 #define int long long
6 const int N = 3e5+5 , mod = 998244353;
7 int n , sum[N][55] , fa[N][20] , dep[N] , logN;
8 vector<int> e[N];
9
10 int read(){...}

```

```

11
12 void dfs( int x ){
13     for( auto v : e[x] ){
14         if( v == fa[x][0] ) continue;
15         dep[v] = dep[x] + 1 , fa[v][0] = x;
16         for( int i = 1 , val = 1; i <= 50 ; i ++ )
17             val = val * dep[v]% mod , sum[v][i] = (val + sum[x][i]) %
                mod;
18         for( int i = 1 ; i <= logN ; i ++ )
19             fa[v][i] = fa[ fa[v][i-1] ][i-1];
20         dfs(v);
21     }
22 }
23
24 int lca( int x , int y ){...}
25
26 int32_t main(){
27     n = read() , logN = (int)log2(n)+1;
28     for( int i = 2 , u , v ; i <= n ; i ++ )
29         u = read() , v = read() , e[u].push_back(v) , e[v].push_back(u
        );
30     dep[1] = 0;
31     dfs( 1 );
32     for( int m = read() , u , v , k , t ; m ; m -- ){
33         u = read() , v = read() , k = read() , t = lca( u , v );
34         cout << (sum[u][k]+sum[v][k]-sum[t][k]-sum[fa[t][0]][k]+2*mod)
            %mod << "\n";
35     }
36     return 0;
37 }

```

如果是边权,  $x,y$  路径上的和为  $sum[x]+sum[y]-2*sum[lca]$

```

1 //LOJ 10134 树上前缀和 边
2 const int N = 1e4+5;
3
4 int n , m , sum[N] , logN , dep[N] , fa[N][15];
5 vector<pair<int,int>> e[N];
6
7 int read(){...}

```

```

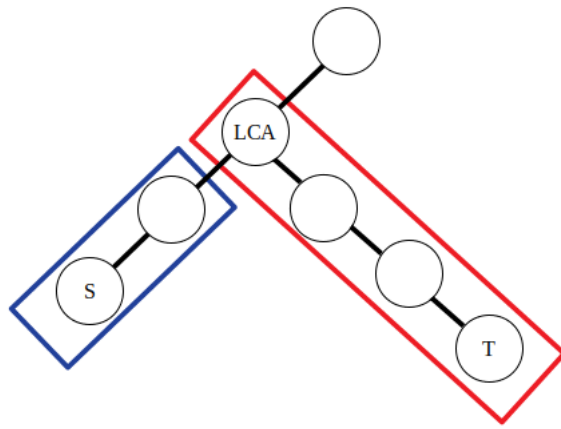
8
9 void dfs( int x ){ // 这里和 lca 极其类似，就是在多维护了一个前缀和
10     for( auto [v,w] : e[x] ){
11         if( dep[v] ) continue;
12         dep[v] = dep[x] + 1 , fa[v][0] = x , sum[v] = sum[x] + w;
13         for( int i = 1 ; i <= logN ; i ++ )
14             fa[v][i] = fa[ fa[v][i-1] ][ i-1 ];
15         dfs(v);
16     }
17 }
18
19 int lca( int x , int y ){...} // 这里就是 lca 的板子
20
21 int main(){
22     n = read() , m = read() , logN = (int)log2(n)+1;
23     for( int i = 2 , u , v , w ; i <= n ; i ++ )
24         u = read() , v = read() , w = read() , e[u].push_back( {v,w} )
25         , e[v].push_back( {u,w} );
26     dep[1] = 1 , dfs(1);
27     for( int u , v ; m ; m -- ){
28         u = read() , v = read();
29         cout << sum[u] + sum[v] - 2 * sum[ lca(u,v) ] << "\n";
30     }
31     return 0;
32 }

```

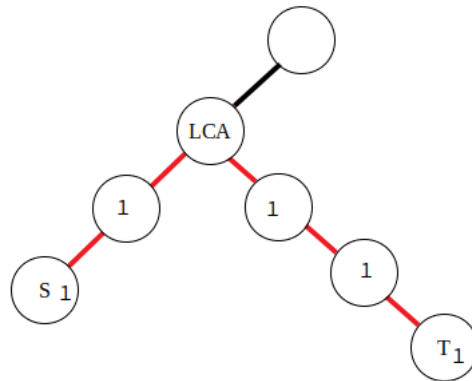
### 3.5.6 树上差分

树上差分就是对树上某一段路径进行差分操作, 树上差分分为**点差分**和**边差分**

这里的差分数组用 $d[i]$ 表示, 其值是 $a[fa[i]]-a[i]$  点差分



点差分实际上是要对链分成两条链来操作, 如果要对  $(S, T)$  路径上点权加  $p$  则要  $d[s] += p$  ,  $d[t] += p$  ,  $d[lca] -= p$  ,  $d[fa[lca]] -= p$   
边差分



因为直接对边差分比较困难, 所以要把边权移动到边上的子节点上, 如果要对  $(S, T)$  路径上边权加  $p$ , 则要  $d[s] += p$  ,  $d[t] += p$  ,  $d[lca] -= 2 * p$

```

1 // Luogu P3128 边差分模板
2 #include<bits/stdc++.h>
3 using namespace std;
4
5 const int N = 5e4+5;
6 int n , m , fa[N][30] , logN , dep[N] , p[N] , res;
7 vector<int> e[N];
8
9 int read(){...}
10

```



```

11 void dfs( int x ) {...} // 和 lca 的 dfs 完全相同
12
13 int lca( int x , int y ){...} // lca
14
15 void getRes( int x ){
16     for( auto v : e[x] ){
17         if( fa[x][0] == v ) continue;
18         getRes(v) , p[x] += p[v];
19     }
20     res = max( res , p[x] );
21 }
22
23 int32_t main(){
24     n = read() , m = read() , logN = log2(n)+1;
25     for( int u , v , i = 1 ; i < n ; i ++ )
26         u = read() , v = read() , e[u].push_back(v) , e[v].push_back(u);
27     dep[1] = 1;
28     dfs( 1 );
29     for( int i = 1 , u , v , d ; i <= m ; i ++ ){
30         u = read() , v = read() , d = lca( u , v );
31         p[u] ++ , p[v] ++ , p[d] -- , p[ fa[d][0] ] --;
32     }
33     getRes( 1 );
34     cout << res << "\n";
35 }

```

## 3.6 有向图连通性

### 3.6.1 强连通分量

**连通**在有向图中存在  $u$  到  $v$  的路径，则称  $u$  可达  $v$ 。如果  $u, v$  互相可达，则  $u, v$  连通。

**强连通**有向图  $G$  强连通指  $G$  中任意两个结点连通。

**强联通分量**有向图的极大强连通子图。

#### DFS 生成树

在有向图上进行 DFS 会形成森林。DFS 会形成 4 种边。

1. Tree Edge, 树边
2. Back Edge, 返祖边, 指向祖先结点的边

3. Cross Edge, 衡叉边, 指向搜索过程中已经访问过的结点, 但是这个结点并不是祖先节点

4. Forward Edge, 前向边, 指向子孙节点的边

如果结点  $u$  是某个强连通分量在搜索树中遇到的第一个结点, 那么这个强连通分量的其余结点肯定是在搜索树中以  $u$  为根的子树中。结点  $u$  被称为这个强连通分量的根。

### Tarjan

```

1 int cnt = 0, sc = 0; // cnt 记录 dfs 序 , sc 记录 强连通分量编号
2 stack<int> stk;
3 vector<int> dfn, low, inStk, scc, capacity;
4 // low[i] 点i所在子树的节点经过最多一条非树边能到达的结点中最小的dfs序
5 void tarjan(int x) {
6     low[x] = dfn[x] = ++cnt;
7     inStk[x] = 1, stk.push(x);
8     for (auto y: e[x]) {
9         if (!dfn[y])
10             tarjan(y), low[x] = min(low[x], low[y]);
11         else if (inStk[y])
12             low[x] = min(low[x], dfn[y]);
13     }
14     if (low[x] == dfn[x]) {
15         sc++, capacity.push_back(0);
16         for (int y; true;) {
17             y = stk.top(), stk.pop();
18             inStk[y] = 0, scc[y] = sc, capacity[sc]++;
19             if (x == y) break;
20         }
21     }
22 }
23
24 // 因为有向图搜索会形成森林
25 for (int i = 1; i <= n; i++)
26     if (!dfn[i]) tarjan(i);

```

### 缩点

```

1 for( int x = 1 ; x <= n ; x ++ )
2     for( auto y : e[x] )
3         if( scc[x] != scc[y] ) g[scc[x]].push_back( scc[y] );

```

## 3.7 无向图连通性

### 3.7.1 割点

若对于  $x \in V$ , 从图中删除节点  $x$  以及所有  $x$  链接的边后,  $G$  分裂成两个或两个以上个不相连的子图, 则称  $x$  为  $G$  的**割点**

#### 割点判定法则

若  $x$  不是搜索树的根节点, 则  $x$  是割点当且仅当搜索树上存在一个  $x$  的子节点  $y$  满足  $dfn[x] \leq low[y]$ .

特别的, 若  $x$  是搜索树的根节点, 则  $x$  是割点的条件当且仅当搜索树上存在至少两个子节点满足.

```

1  vector<vector<int>>> e;
2
3  int cnt = 0;
4  vector<int> dfn, low;
5  vector<int> cut; // 储存所有的割点
6
7  void tarjan( int p , bool root = true ){
8      int tot = 0;
9      low[p] = dfn[p] = ++ cnt;
10     for( auto q : e[p] ){
11         if( !dfn[q] ){
12             tarjan( q , false );
13             low[p] = min( low[p] , low[q] );
14             tot += ( low[q] >= dfn[p] ); // 统计满足条件的子节点数
15         }else low[p] = min( low[p] , dfn[q] );
16     }
17     if( tot > root ) cut.push_back(p);
18     return ;
19 }
```

### 3.7.2 桥

若对于  $e \in E$ , 从图中删除边  $e$  之后,  $G$  分裂成两个不相连的子图, 则称  $e$  为  $G$  的**桥**或**割边**。

#### 割边判定法则

无向边  $(x, y)$  是桥, 当且仅当搜索树上存在  $x$  的一个子节点  $y$ , 满足  $dfn[x] \leq low[y]$ .

桥一定是搜索树中的边, 一个简单环中的边一定都不是桥。

```

1  vector<pii> bridges;
2  vector<vi> e;
```

```

3 vi dfn, low, fa;
4 int cnt;
5
6 void tarjan(int x) {
7     low[x] = dfn[x] = ++cnt;
8     for (auto y: e[x]) {
9         if (!dfn[y]) {
10             fa[y] = x, tarjan(y);
11             low[x] = min(low[x], low[y]);
12             if (low[y] > dfn[x])
13                 bridges.emplace_back(x, y);
14         } else if (fa[x] != y)
15             low[x] = min(low[x], dfn[y]);
16     }
17     return;
18 }

```

## 3.8 图论杂项

### 3.8.1 判断简单无向图图

根据图中的每个点的度可以判断，这个图是不是一个简单无向图（没有重边和自环的无向图）

#### Havel-Hakimi 定理

1. 对当前数列排序，使其呈递减
2. 从  $S[2]$  开始对其后  $S[1]$  个数字-1
3. 一直循环直到当前序列出现负数（即不可简单图化的情况）或者当前序列全为 0（可简单图化）时退出。

```

1 //NC-contest-38105-K
2 int read() {...}
3 priority_queue<int> q; vector<int> ve;
4 int32_t main() {
5     int n = read();
6     for( int i = 1 , x ; i <= n ; i ++ ){
7         x = read() , q.push(x);
8         if( x >= n ) // 如果度数大于 n 一定存在重边或自环
9             cout << "NO\n" , exit(0);

```

```

10     }
11     while( 1 ){
12         int k = q.top(); q.pop();
13         if( k == 0 )
14             cout << "YES\n" , exit(0);
15         ve.clear();
16         for( int x ; k ; k -- ){
17             x = q.top() - 1 , q.pop();
18             if( x < 0 )
19                 cout << "NO\n" , exit(0);
20             ve.push_back(x);
21         }
22         for( auto it : ve ) q.push(it);
23     }
24     return 0;
25 }

```

### 3.8.2 2-SAT

SAT 是是适定性 (Satisfiability) 问题的简称。一般形式为  $k$ -适定性问题, 简称  $k$ -SAT。而当  $k > 2$  时该问题为 NP 完全的。

#### 定义

有  $n$  个布尔变量  $x_1 \sim x_n$ , 另有  $m$  个需要满足的条件, 每个条件的形式都是「 $x_i$  为 true/false 或  $x_j$  为 true/false」。比如「 $x_1$  为真或  $x_3$  为假」、「 $x_7$  为假或  $x_2$  为假」。

$$(x_i x_j)(\neg x_i x_j) \cdots$$

注意这里的或为**排斥或**

2-SAT 问题的目标是给每个变量赋值使得所有条件得到满足。

#### Tarjan SCC 缩点

把每个变量拆成两个点,  $X(True)$  和  $X(False)$ 。比如现在有一个要求  $X|Y = True$ , 则把  $X(False)$  到  $Y(True)$  连一条边, 把  $X(True)$  到  $Y(False)$  连一条边。连出来的图是对称的, 然后跑一遍 Tarjan, 如果存在某个变量的两个点在同一个强连通分量中的情况, 则无解, 否则有解。

构造方案时, 我们可以通过缩点后的拓扑序确定变量的值。如果变量  $x$  的拓扑序在  $\neg x$  之后, 则取  $x$  为真, 反之取  $x$  为假。注意 Tarjan 求得的 SCC 的编号相同与反拓扑序。

```

1 // luogu P4782
2 int32_t main() {
3     int n, m, N;
4     cin >> n >> m, N = n * 2 + 2;

```

```

5     e.resize(N);
6     dfn = inStk = low = scc = vector<int>(N);
7     capacity.push_back(0);
8     for (int i, a, j, b; m; m--) {
9         cin >> i >> a >> j >> b;
10        e[2 * i + (a ^ 1)].push_back(2 * j + b);
11        e[2 * j + (b ^ 1)].push_back(2 * i + a);
12    }
13    for (int i = 2; i <= n * 2 + 1; i++)
14        if (!dfn[i])
15            tarjan(i);
16    vector<int> res(n + 1);
17    for (int i = 1; i <= n; i++) {
18        if (scc[i * 2] == scc[i * 2 + 1])
19            cout << "IMPOSSIBLE\n", exit(0);
20        else if (scc[i * 2] > scc[i * 2 + 1])
21            res[i] = 1;
22    }
23    cout << "POSSIBLE\n";
24    for (int i = 1; i <= n; i++)
25        cout << res[i] << " ";
26    cout << "\n";
27    return 0;
28 }

```

## 3.9 二分图

### 3.9.1 二分图

#### 定义

给一张无向图，可以把点分成两个不相交的非空集合，并且在同一集合的点之间没有边相连，那么称这张无向图为一个二分图。

#### 二分图的判定

一张无向图是二分图，当且仅当图中不存在奇环。

```

1 vector<vector<int>>> e;
2 vector<int> v; // 会把所有的点染成 1 2 两种颜色
3 bool flag;
4

```

```

5 void dfs( int x , int color ){
6     v[x] = color;
7     for( auto y : e[x] ){
8         if( v[y] == 0 ) dfs( y , 3 - color );
9         else if( v[y] == color ){
10             flag = false;
11             return ;
12         }
13     }
14 }
15
16 bool check(){
17     flag = true;
18     for( int i = 1 ; flag and i <= n ; i ++ ){
19         if( v[i] ) continue;
20         dfs( i , 1 );
21     }
22     return flag;
23 }

```

### 3.9.2 二分图最大匹配

“任意两条边都没有公共端点”的边集合被称为一组匹配。

在二分图中包含边数最多的一组匹配被称为二分图的最大匹配。

对于任意一组匹配  $S$  ( $S$  是一个边集), 属于  $S$  的边叫匹配边, 匹配边的端点叫匹配点。

如果在二分图中存在连接两个非匹配点的路径  $path$ , 则称  $path$  是  $S$  的增广路。

增广路存在以下性质:

1. 长度是奇数
2. 路径上第 1, 3, 5, ... 是非匹配边, 第 2, 4, 6, ... 是匹配边

所以对于一组匹配, 把增广路上的边的状态全部取反, 得到新的边集合  $S'$ , 则  $S'$  也是一组匹配, 且匹配变数多一。

所以二分图的一组最大匹配  $S$ , 当且仅当二分图中不存在  $S$  的增广路。

#### 匈牙利算法

##### 算法流程

1. 设  $S = \emptyset$
2. 求增广路  $path$ , 把路径上边的状态取反得到新的匹配  $S'$

## 3. 重复第 2 步直到没有增广路

代码实现采用深搜，从  $x$  出发寻找增广路，并且还回溯时把状态取反。

$N$  个点  $M$  条的二分图，复杂度  $O(NM)$

```
1 // luogu P3386
2 // 给定一个二分图，其左侧点  $n$  个，右侧点的个数为  $m$ ，边数为  $k$ ，求其最大匹配的边数。
3 // 左侧点编号  $[1, n]$ ，右侧点编号  $[1, m]$ 。
4 #include <bits/stdc++.h>
5
6 using namespace std;
7
8 int n, m, k; // 左右两个集合的元素数量
9 vector<vector<int>>> e;
10 vector<int> p; // 当前右侧集合对于左侧的元素
11 vector<bool> vis; // 右侧元素是否已经被访问过
12
13 bool match(int x) {
14     for (auto y: e[x]) {
15         if (vis[y]) continue;
16         vis[y] = 1;
17         if (p[y] == 0 or match(p[y])) { // 暂未匹配或原来匹配的左侧元素可以找到新的匹配
18             p[y] = x;
19             return true;
20         }
21     }
22     return false;
23 }
24
25 int Hungarian() {
26     int cnt = 0;
27     p = vector<int>(n + m + 1);
28     for (int i = 1; i <= n; i++) { // 枚举左侧元素
29         vis = vector<bool>(n + m + 1);
30         if (match(i)) cnt++;
31     }
32     return cnt;
33 }
```



```

34
35 int32_t main() {
36     ios::sync_with_stdio(false), cin.tie(nullptr);
37     cin >> n >> m >> k;
38     e = vector<vector<int>>>(n + m + 1);
39
40     for (int x, y; k; k--) {
41         cin >> x >> y;
42         y += n; // 为了方便表示把右侧点映射为 [n+1,n+m]
43         e[x].push_back(y), e[y].push_back(x);
44     }
45     cout << Hungarian() << "\n";
46     return 0;
47 }

```

### 3.9.3 二分图的最小的点覆盖

给一张二分图，求最小点集  $S$ ，使得图中任意一条边都至少有一个端点属于  $S$ 。

#### König 定理

二分图最小点覆盖包含的点数等于二分图最大匹配包含的边数。

构造方法

1. 求出二分图的最大匹配
2. 从左部每个非匹配点出发，再执行一次 DFS 求增广路的过程（一定会失败），标记访问过的所有点
3. 取左部未被标记的点、右部被标记的点，就得到了最小点覆盖。

### 3.9.4 二分图的最大独立集

给一张无向图，图的独立集就是任意两点之间没有边相连的点集，包含点数最多的独立集就是图的最大独立集。

任意两点之间都有一条边相连的子图被称作无向图的团，点数最多团就是图的最大团。

对于一般的无向图，最大团、最大独立集是 NP 完全问题。

#### 定理

无向图的最大团就是补图的最大独立集

#### 定理

$G$  是  $n$  个点的二分图， $G$  的最大独立集大小等于  $n$  减去最大匹配数。

对于二分图去掉最小点覆盖，剩余的点就构成了二分图的最大独立集。



## 第四章 数学知识

### 4.1 数论

#### 4.1.1 整除

定义：若整数  $b$  除以非零整数  $a$ ，商为整数且余数为零我们就说  $b$  能被  $a$  整除，或  $a$  整除  $b$  记作  $a|b$

性质

1. 传递性，若  $a|b, b|c$ ，则  $a|c$
2. 组合性，若  $a|b, a|c$  则对于任意整数  $m, n$  均满足  $a|mb + nc$
3. 自反性，对于任意的  $n$ ，均有  $n|n$
4. 对称性，若  $a|b, b|a$  则  $a = b$

#### 4.1.2 约数

定义若整数  $n$  除以整数  $x$  的余数为 0，即  $d$  能整除  $n$ ，则称  $d$  是  $n$  的约数， $n$  是  $d$  的倍数，记为  $d|n$  算数基本定理由算数基本定理得正整数  $N$  可以写作  $N = p_1^{C_1} \times p_2^{C_2} \times p_3^{C_3} \cdots \times p_m^{C_m}$

分解质因数

分解成  $p_1 \times p_2 \times p_3 \times \cdots \times p_n$  这种形式

```
1 vector< int > factorize( int x ){
2     vector<int> ans;
3     for( int i = 2 ; i * i <= x ; i ++){
4         while( x % i == 0 )
5             ans.push_back(i) , x /= i;
6     }
7     if( x > 1 ) ans.push_back(x);
8     return ans;
9 }
```

分解成  $p_1^{k_1} \times p_2^{k_2} \times p_3^{k_3} \times \cdots \times p_n^{k_n}$

```

1 vector< pair<int,int> > factorize( int x ){
2     vector<pair<int,int>> ans;
3     for( int i = 2 , cnt ; i * i <= x ; i ++){
4         if( x % i ) continue;
5         cnt = 0;
6         while( x % i == 0 ) cnt ++ , x /= i;
7         ans.push_back( { i , cnt } );
8     }
9     if( x > 1 ) ans.push_back( { x , 1 } );
10    return ans;
11 }

```

$N$  的正约数个数为 ( $\Pi$  是连乘积的符号, 类似  $\sum$ )

$$(c_1 + 1) \times (c_2 + 1) \times \cdots (c_m + 1) = \Pi_{i=1}^m (c_i + 1)$$

$N$  的所有正约数和为

$$(1 + p_1 + p_1^2 + \cdots + p_1^{c_1}) \times \cdots \times (1 + p_m + p_m^2 + \cdots + p_m^{c_m}) = \prod_{i=1}^m \left( \sum_{j=0}^{c_i} (p_i)^j \right)$$

### 4.1.3 GCD 和 LCM

性质  $a \times b = \gcd(a, b) \times (a, b)$

通过性质可以得到最小公倍数的求法就是

```

1 int lcm( int x , int y ){
2     return a / gcd( x , y ) * b;
3 }

```

最大公倍数的求法有更相减损术和辗转相除法

```

1 int gcd( int x , int y ){ // 更相减损术
2     while( x != y ){
3         if( x > y ) x -= y;
4         else y -= x;
5     }
6     return x;
7 }
8
9 int gcd( int x , int y ){ // 辗转相除法
10    return b ? gcd( b , a % b ) : a ;
11 }

```

```

12
13 // 还有一种是直接调用库函数
14 __gcd( a , b );

```

一般情况下直接用库函数，库函数的实现是辗转相除法，如果遇到高精度的话（高精度取模分困难）可以用更相减损术来代替

**定理**对于斐波那契数列  $Feb_i$  有  $Feb_{gcd(a,b)} = gcd(Feb_a, Feb_b)$

#### 4.1.4 质数

判断质数

```

1 bool isPrime( int x ){
2     for( int i = 1 ; i * i <= x ; i ++ )
3         if( x % i == 0 ) return 0;
4     return 1;
5 }

```

埃式筛

```

1 vector< int > prime;
2 bitset<N>notPrime;//不是素数
3
4 void getPrimes( int n ){
5     notPrime[1] = notPrime[0] = 1;
6     for( int i = 2 ; i <= n ; i ++ ){
7         if( notPrime[i] ) continue;
8         prime.push_back(i);
9         for( int j = i * 2 ; j <= n ; j += i )
10             notPrime[j] = 1;
11     }
12 }
13
14 // 如果不需要 prime 数组的话可以优化成下面的代码
15 bitset<N>notPrime;//不是素数
16
17 void getPrimes( int n ){
18     notPrime[1] = notPrime[0] = 1;
19     for( int i = 2 ; i * i <= n ; i ++ ){
20         if( notPrime[i] ) continue;
21         for( int j = i * 2 ; j <= n ; j += i )
22             notPrime[j] = 1;

```

```

23     }
24 }

```

### 欧拉筛

```

1  vector< int > prime;
2  bool notPrime[N];; // 不是素数
3
4  void getPrimes( int n ){
5      notPrime[1] = notPrime[0] = 1;
6      for( int i = 2 ; i <= n ; i ++ ){
7          if( !notPrime[i] ) prime.push_back(i);
8          for( auto it : prime ){
9              if( it * i > n ) break;
10             notPrime[ it * i ] = 1;
11             if( i % it == 0 ) break;
12         }
13     }
14 }

```

### 证明质数有无限个

反证法假设数是  $n$  个，每个素数是  $p_i$ ，令  $P = \prod_{i=1}^n p_i + 1$

因为任何一个数都可以分解成多个质数相乘

所以  $P$  除以任何一个质数都余 1，显然  $P$  就也是一个质数，与假设矛盾，所以假设错误  
所以质数是无限个

### 性质 2

设  $\pi(n)$  为不超过  $n$  的质数个数，则  $\pi(n) \approx \frac{n}{\ln n}$

## 4.1.5 逆元

### 费马小定理

$a^{p-1} \equiv 1(\text{mod } p)$ , 其中  $p$  为素数, 所以  $aa^{p-2} \equiv 1(\text{mod } p)$

```

1  int inv( int x ) {return pow( x , p - 2 );}

```

### $O(n)$ 递推

```

1  const int N = 1005;
2  int inv[N] = {};
3  void invers(int n,int mod){
4      inv[1] = 1;
5      for(int i = 2;i <= n;i ++) inv[i] = (p-p/i) * inv[p%i] % p;
6      return ;

```

7 }

### 4.1.6 扩展欧几里得

#### 裴蜀定理

设  $a, b$  是不全为零的整数, 则存在整数  $x, y$ , 使得  $ax + by = \gcd(a, b)$

```

1 int exgcd( int a , int b , int & x , int & y ){
2     if( b == 0 ) { x = 1 , y = 0 ; return a;}
3     int d = exgcd( b , a%b , x , y );
4     int z = x ; x = y ; y = z - y * (a / b);
5     return d;
6 }
```

#### 丢番图方程

$$ax + by = c$$

定义变量  $d, x_0, y_0$ , 调用  $d = \text{exgcd}(a, b, x_0, y_0)$ 。对于方程的特解为

$$(x = \frac{c}{d}x_0, y = \frac{c}{d}y_0)$$

对于方程的通解为

$$(x = \frac{c}{d}x_0 + k\frac{b}{d}, y = \frac{c}{d}y_0 - k\frac{a}{d}), k \in Z$$

#### 线性同余方程

$$a \times x \equiv b \pmod{m}$$

线性同余方程等价于  $a \times x - b$  是  $m$  的倍数, 设为  $-y$  倍, 方程可改写为丢番图方程  $a \times x + m \times y = b$

线性同余方程有解的充要条件  $\gcd(a, m) | b$

在有解时用扩偶求得  $x_0, y_0$  满足  $a \times x_0 + m \times y_0 = \gcd(a, m)$ , 则方程的特解  $x = x_0 \times \frac{b}{\gcd(a, m)}$

通解是  $x = x_0 \times \frac{b}{\gcd(a, m)} + k \times \frac{m}{\gcd(a, m)}, k \in$

```

1 int calc(int a, int b, int m) {
2     int x, y, d;
3     d = exgcd(a, m, x, y);
4     if (b % d) return -1;
5     return x * b / d;
6 }
```

#### 扩展欧几里得求逆元

本质上是解同余方程  $a \times a^{-1} \equiv 1 \pmod{m}$

```

1 int inv(int a, int m) {
2     int x, y, d;
```

```

3      d = exgcd(a, m, x, y);
4      if (d != 1) return -1;
5      return (x % m + m) % m;
6  }
```

### 线性同余方程组 (中国剩余定理)

设  $m_1, m_2, \dots, m_n$  是两两互质的整数。 $m = \prod_{i=1}^n m_i$ ,  $M_i = \frac{m}{m_i}$ ,  $t_i$  是线性同余方程组  $M_i t_i \equiv 1 \pmod{m}$  的一个解, 对于任意的  $n$  个整数  $a_1, a_2, \dots, a_n$ , 方程组

$$\begin{cases} x \equiv a_1 \pmod{m_1} \\ x \equiv a_2 \pmod{m_2} \\ \vdots \\ x \equiv a_n \pmod{m_n} \end{cases}$$

有整数解, 解为  $x = \sum_{i=1}^n a_i M_i t_i \pmod{m}$

```

1 int CRT(int n, vector<int> a, vector<int> m) {
2     int mm = 1, ans = 0;
3     for (int i = 1; i <= n; i++) mm = mm * m[i];
4     for (int i = 1; i <= n; i++) {
5         int M = mm / m[i], t, y;
6         exgcd(M, m[i], t, y); // t * M % m[i] = 1;
7         ans = (ans + a[i] * M * t % mm) % mm;
8     }
9     return ans;
10 }
```

### 高次同余方程 (BSGS)

$a^x \equiv b \pmod{p}$ , 要求  $a, p$  互质。求非负整数  $x$ 。复杂度  $O(\sqrt{p})$

```

1 int BSGS(int a, int b, int p) {
2     map<int, int> hash;
3     b %= p;
4     int t = sqrt(p) + 1;
5     for (int j = 0, val; j < t; j++) {
6         val = b * power(a, j, p) % p;
7         hash[val] = j;
8     }
9     a = power(a, t, p);
10    if (a == 0) return b == 0 ? 1 : -1;
11    for (int i = 0, val, j; i <= t; i++) {
12        val = power(a, i, p);
```



```

13         j = hash.find(val) == hash.end() ? -1 : hash[val];
14         if (j >= 0 && i * t - j >= 0) return i * t - j;
15     }
16     return -1;
17 }

```

## 4.2 数学杂项

### 4.2.1 二维向量的叉积

$$\vec{A} \times \vec{B} = |A||B| \cos(\alpha) = a_x \times b_y - a_y \times b_x$$

虽然叉积的结果是一个标量，但是叉积是有正负的，正负取决于向量夹角的大小。

**应用：判断点是否在三角形的内部**对于三角形  $abc$  和一个点  $o$ ， $\vec{ao} \times \vec{ab}$  的正负表示了点  $o$  在线  $ab$  的左侧还是右侧。只要按照顺时针方向（或逆时针方向）判断点  $o$  在三条直线的同一侧，既可以判断点在三角形的内部。

```

1 class Triangle{
2     typedef std::pair<int,int> Vector;
3 private:
4     Vector getVector( int x , int y , int a , int b ){
5         return std::pair{ a - x , b - y };
6     }
7     bool product( Vector a , Vector b ){
8         return ( a.first * b.second - a.second * b.first ) > 0;
9     }
10 public:
11     int ax , ay , bx ,by , cx , cy;
12     Vector ab , bc , ca;
13     Triangle( int ax , int ay , int bx , int by , int cx , int cy ){
14         ax(ax) , ay(ay) , bx(bx) , by(by) , cx(cx) , cy(cy) ,
15         ab( getVector( ax , ay , bx , by ) ),
16         bc( getVector( bx , by , cx , cy ) ),
17         ca( getVector( cx , cy , ax , ay ) ){};
18     Triangle(){};
19     bool isInTriangle( int x , int y ){
20         Vector ao = getVector( ax , ay , x , y );
21         Vector bo = getVector( bx , by , x , y );
22         Vector co = getVector( cx , cy , x , y );

```

```

23         bool f1 = product( ao , ab ) , f2 = product( bo , bc ) , f3 =
                product( co , ca );
24         return ( f1 == f2 && f2 == f3 );
25     }
26 };

```

## 4.3 组合数学

### 4.3.1 公式

排列数

$$A_n^n = \frac{n!}{(n-n)!} = \frac{n!}{0!} = n!$$

组合数

$$C_m^n = \frac{m!}{(m-n)! \times n!}$$

组合数性质

1.  $C_n^m = C_n^{n-m}$
2.  $C_n^m = C_{n-1}^m + C_{n-1}^{m-1}$
3.  $C_n^0 + C_n^1 + \cdots + C_n^n = 2^n$
4.  $C_n^0 + C_n^2 + C_n^4 + \cdots = C_n^1 + C_n^3 + C_n^5 \cdots = 2^{n-1}$
5.  $C_n^m = \frac{n-m+1}{n} \times C_n^{m-1}$

### 4.3.2 组合数计算

```

1 // 暴力计算组合数
2 int C(int x, int y) { // x 中选 y 个
3     y = min(y, x - y);
4     int res = 1;
5     for (int i = x, j = 1; j <= y; i--, j++)
6         res = res * i / j;
7     return res;
8 }
9
10 // 加法递推 O(n^2)
11 for( int i = 0 ; i <= n ; i ++ ){
12     c[i][0] = 1;

```

```

13     for( int j = 1 ; j <= i ; j ++ )
14         c[i][j] = c[i-1][j] + c[i-1][j-1];
15 }
16 // 乘法递推O(n)
17 c[0] = 1;
18 for( int i = 1 ; i * 2 <= n ; i ++ )
19     c[i] = c[n-i] = ( n-i+1 ) * c[i-1] / i;
20
21 // 任意组合数
22 #define int long long
23 const int N = 5e5+5 , mod = 1e9+7;
24 int fact[N] , invFact[N];
25
26 int power(int x,int y){...} // 快速幂
27 int inv( int x ){...} // 求逆元, 一般是费马小定理
28
29 int A( int x , int y ){ // x 中选 y 排序
30     return fact[x] * invFact[x-y] % mod;
31 }
32
33 int C( int x , int y ){ // x 中选 y 个
34     return fact[x] * invFact[x-y] % mod * invFact[y] % mod;
35 }
36
37 void init(){
38     fact[0] = 1 , invFact[0] = inv(1);
39     for( int i = 1 ; i < N ; i ++ )
40         fact[i] = fact[i-1] * i % mod , invFact[i] = inv(fact[i]);
41 }

```

### 4.3.3 公式杂项

$$1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

## 4.4 线性代数

### 4.4.1 矩阵加速递推

```
1 struct matrix {
2     static constexpr int mod = 1e9 + 7;
3     int x, y;
4     vector<vector<int>> v;
5
6     matrix() {}
7
8     matrix(int x, int y) : x(x), y(y) {
9         v = vector<vector<int>>(x + 1, vector<int>(y + 1, 0));
10    }
11
12    void I() { // 单位化
13        y = x;
14        v = vector<vector<int>>(x + 1, vector<int>(x + 1, 0));
15        for (int i = 1; i <= x; i++) v[i][i] = 1;
16        return;
17    }
18
19    void display() { // 打印
20        for (int i = 1; i <= x; i++)
21            for (int j = 1; j <= y; j++)
22                cout << v[i][j] << " \n"[j == y];
23        return;
24    }
25
26    friend matrix operator*(const matrix &a, const matrix &b) { // 乘法
27        assert(a.y == b.x);
28        matrix ans(a.x, b.y);
29        for (int i = 1; i <= a.x; i++)
30            for (int j = 1; j <= b.y; j++)
31                for (int k = 1; k <= a.y; k++)
32                    ans.v[i][j] = (ans.v[i][j] + a.v[i][k] * b.v[k][j]
33                                   ) % mod;
34        return ans;
35    }
36
37    friend matrix operator^(matrix x, int y) { // 快速幂
38        assert(x.x == x.y);
```

```

38     matrix ans(x.x , x.y);
39     ans.I();//注意一定要先单位化
40     while( y ){
41         if( y&1 ) ans = ans*x;
42         x = x * x , y >>= 1;
43     }
44     return ans;
45 }
46 };

```

### 例题 Luogo P1939

已知数列  $a$ ，满足

$$a_i = \begin{cases} 1 & i \in \{1, 2, 3\} \\ a_{i-1} + a_{i-3} & i \geq 4 \end{cases}$$

求数列第  $n$  项对  $10^9 + 7$  取模

设计状态阵  $mat_i = \begin{bmatrix} a_i \\ a_{i-1} \\ a_{i-2} \end{bmatrix}$ ，则  $mat_{i+1} = \begin{bmatrix} a_{i+1} \\ a_i \\ a_{i-1} \end{bmatrix} = \begin{bmatrix} a_i + a_{i-2} \\ a_i \\ a_{i-1} \end{bmatrix}$

可以用待定系数法，加对应项相等解出转移矩阵

$$\begin{bmatrix} 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\text{则 } mat_{3+n} = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}^n \times mat_3$$

## 4.5 离散数学

## 4.6 计算几何

### 4.6.1 基础模板

```

1  /* 板子来自 Pecco 采取以下原则
2   * 写全局函数，而非类方法，结构体只存储数据
3   * 每个函数标注以来那些函数，且尽量减少依赖
4   * 用较为简略的名字，同时传值而非 const 引用
5   * */
6
7  #define mp make_pair
8  using db = double; //精度不够时可换 long double

```

```

9 // 几何对象
10 struct Point { // 点
11     db x, y;
12
13     Point(db x = 0, db y = 0) : x(x), y(y) {};
14 };
15
16 using Vec = Point; // 向量
17 struct Line { // 直线 (点向式)
18     Point P;
19     Vec v;
20
21     Line(Point P, Vec v) : P(P), v(v) {};
22 };
23
24 struct Seg { // 线段(存两个端点)
25     Point A, B;
26
27     Seg(Point A, Point B) : A(A), B(B) {};
28 };
29
30 struct Circle { // 圆(存圆心和半径)
31     Point O;
32     db r;
33
34     Circle(Point O, db r) : O(O), r(r) {};
35 };
36
37 // 常用常量
38 const Point O(0, 0); // 原点
39 const Line Ox(0, Vec(1, 0)), Oy(0, Vec(0, 1)); // 坐标轴
40 const double pi = M_PI, eps = 1e-9; // pi 精度最高的是acosl(-1)
41
42 // 浮点比较
43 bool eq(db a, db b) { return abs(a - b) < eps; } // ==
44 bool gt(db a, db b) { return a - b > eps; } // >
45 bool lt(db a, db b) { return a - b < -eps; } // <
46 bool ge(db a, db b) { return a - b > -eps; } // >=

```

```

47 bool le(db a, db b) { return a - b < eps; }      // <=
48
49 // 基础操作
50 Vec r90a(Vec v) { return Vec(-v.y, v.x); }// 顺时针旋转 90 度
51 Vec r90c(Vec v) { return Vec(v.y, -v.x); }// 逆时针旋转 90 度
52 // 向量加法
53 Vec operator+(Vec u, Vec v) { return Vec(u.x + v.x, u.y + v.y); }
54
55 // 向量减法
56 Vec operator-(Vec u, Vec v) { return Vec(u.x - v.x, u.y - v.y); }
57
58 // 数乘
59 Vec operator*(db k, Vec v) { return Vec(k * v.x, k * v.y); }
60
61 // 点乘
62 db operator*(Vec u, Vec v) { return u.x * v.x + u.y * v.y; }
63
64 // 叉乘
65 db operator^(Vec u, Vec v) { return u.x * v.y - u.y * v.x; }
66
67 db len(Vec v) { return sqrt(v.x * v.x + v.y * v.y); }// 向量长度
68 db slope(Vec v) { return v.y / v.x; }// 斜率
69
70 // 向量相关
71 // 两向量夹角余弦 u->v depends len,V*V
72 double cos_t(Vec u, Vec v) { return u * v / len(u) / len(v); }
73
74 //单位化 depends len
75 Vec norm(Vec v) { return Vec(v.x / len(v), v.y / len(v)); }
76
77 // 与原向量平行且横坐标大于等于 0 的单位向量 depends k*V , len , norm;
78 Vec pnorm(Vec v) {
79     return (v.x < 0 ? -1 : 1) * norm(v);
80     return (v.x < 0 ? -1 : 1) / len(v) * v;// 不依赖 norm
81 }
82
83 // 线段的方向向量 depends V-V // 直线的方向向量直接访问 v
84 Vec dvec(Seg l) { return l.B - l.A; }

```

```

85
86 // 直线相关
87 // 两点式求直线 depends V-V
88 Line line(Point A, Point B) { return Line(A, B - A); }
89
90 // 斜截式求直线
91 Line line(db k, db b) { return Line(Point(0, b), Vec(1, k)); }
92
93 // 点斜式求直线
94 Line line(Point P, db k) { return Line(P, Vec(1, k)); }
95
96 // 线段所在直线 depends V-V
97 Line line(Seg l) { return Line(l.A, l.B - l.A); }
98
99 db at_x(Line l, db x) { // 给定直线的横坐标求纵坐标
100     assert(l.v.x != 0);
101     return l.P.y + (x - l.P.x) * l.v.y / l.v.x;
102 }
103 db at_y(Line l, db y) { // 给定直线的纵坐标求横坐标
104     assert(l.v.y != 0);
105     return l.P.x + (y - l.P.y) * l.v.x / l.v.y;
106 }
107
108 Point pedal(Point P, Line l) { // 点到直线的垂足 depends V-V, V*V, d*V
109     return l.P - (l.P - P) * l.v / (l.v * l.v) * l.v;
110 }
111
112 // 过某点作直线的垂线 depends r90c
113 Line perp(Line l, Point P) { return Line(P, r90c(l.v)); }
114
115 Line bisec(Point P, Vec u, Vec v) { // 角平分线 depends V+V, len, norm
116     return Line(P, norm(u) + norm(v));
117 }
118
119 // 线段相关
120 Point midp(Seg l) { // 线段中点
121     return Point((l.A.x + l.B.x) / 2, (l.A.y + l.B.y) / 2);
122 }

```



```

123 Line perp(Seg l) { // 线段中垂线 depends r90c, V-V, midp
124     return Line(midp(l), r90c(l.B - l.A));
125 }
126
127 // 几何对象之间的关系
128 // 向量是否垂直 depends eq, V*V
129 bool verti(Vec u, Vec v) { return eq(u * v, 0); }
130
131 // 向量是否平行 depends eq, V^V
132 bool paral(Vec u, Vec v) { return eq(u ^ v, 0); }
133
134 // 向量是否与x轴平行 depends eq
135 bool paral_x(Vec v) { return eq(v.y, 0); }
136
137 // 向量是否与y轴平行 depends eq
138 bool paral_y(Vec v) { return eq(v.x, 0); }
139
140 bool on(Point P, Line l) { // 点是否在直线上 depends eq
141     return eq((P.x - l.P.x) * l.v.y, (P.y - l.P.y) * l.v.x);
142 }
143
144 bool on(Point P, Seg l) { // 点是否在线段上 depends eq, len, V-V
145     return eq(len(P - l.A) + len(P - l.B), len(l.A - l.B));
146 }
147
148 bool operator==(Point A, Point B) { // 两点是否重合 depends eq
149     return eq(A.x, B.x) and eq(A.y, B.y);
150 }
151
152 bool operator==(Line a, Line b) { // 两条直线是否重合 depends eq, on(L)
153     return on(a.P, b) and on(a.P + a.v, b);
154 }
155
156 bool operator==(Seg a, Seg b) { // 两条线段是否重合 depends eq, P==P
157     return (a.A == b.A and a.B == b.B) or (a.A == b.B and a.B == b.A);
158 }
159
160 // 以横坐标为第一关键字、纵坐标为第二关键字比较两个点 depends eq, lt

```

```

161 bool operator<(Point A, Point B) {
162     return lt(A.x, B.x) or (eq(A.x, B.x) and lt(A.y, B.y));
163 }
164
165 // 直线与圆是否相切 depends eq, V^V, len
166 bool tangency(Line l, Circle c) {
167     return eq(abs((c.0 ^ l.v) - (l.P ^ l.v)), c.r * len(l.v));
168 }
169
170 // 圆与圆是否相切 depends eq, V-V, len
171 bool tangency(Circle C1, Circle C2) {
172     return eq(len(C1.0 - C2.0), C1.r + C2.r);
173 }
174
175 // 距离
176 // 两点之间的距离 depends len, V-V
177 db dis(Point A, Point B) { return len(A - B); }
178
179 // 点到直线的距离 depends V^V, len
180 db dis(Point P, Line l) {
181     return abs((P ^ l.v) - (l.P ^ l.v)) / len(l.v);
182 }
183
184 // 平行直线间的距离 depends d*V, V^V, len, pnrom
185 db dis(Line a, Line b) {
186     assert(paral(a.v, b.v));
187     return abs((a.P ^ pnorm(a.v)) - (b.P ^ pnorm(b.v)));
188 }
189
190 // 平移和旋转
191 // 平移 depends V+V
192 Line operator+(Line l, Vec v) { return Line(l.P + v, l.v); }
193 Seg operator+(Seg l, Vec v) { return Seg(l.A + v, l.B + v); }
194
195 // 旋转 depends V+V, V-V
196 Point rotate(Point P, db rad) {
197     return Point(cos(rad)*P.x-sin(rad)*P.y,sin(rad)*P.x+cos(rad)*P.y);
198 }

```

```

199 Point rotate(Point P, db rad, Point C) {
200     return C + rotate(P - C, rad);
201 }
202
203 Line rotate(Line l, db rad, Point C = 0) {
204     return Line(rotate(l.P, rad, C), rotate(l.v, rad));
205 }
206
207 Seg rotate(Seg l, db rad, Point C = 0) {
208     return Seg(rotate(l.A, rad, C), rotate(l.B, rad, C));
209 }
210
211 // 对称
212 // 关于点的对称
213 Point reflect(Point A, Point P) {
214     return Point(P.x * 2 - A.x, P.y * 2 - A.y);
215 }
216
217 Line reflect(Line l, Point P) { return Line(reflect(l.P, P), l.v); }
218
219 Seg reflect(Seg l, Point P) {
220     return Seg(reflect(l.A, P), reflect(l.B, P));
221 }
222
223 // 关于直线对称 depends V-V, V*V, d*V , pedal
224 Point reflect(Point A, Line ax) { return reflect(A, pedal(A, ax)); }
225
226 Vec reflect_v(Vec v, Line ax) {
227     return reflect(v, ax) - reflect(0, ax);
228 }
229
230 Line reflect(Line l, Line ax) {
231     return Line(reflect(l.P, ax), reflect_v(l.v, ax));
232 }
233
234 Seg reflect(Seg l, Line ax) {
235     return Seg(reflect(l.A, ax), reflect(l.B, ax));
236 }

```

```

237 // 交点
238 // 直线与直线交点 depends eq, d*V, V*V, V+V, V^V
239 vector <Point> inter(Line a, Line b) {
240     vector <Point> ans;
241     double c = a.v ^ b.v;
242     if (eq(c, 0)) return ans;
243     Vec v = 1 / c * Vec(a.P ^ (a.P + a.v), b.P ^ (b.P + b.v));
244     ans.emplace_back(v * Vec(-b.v.x, a.v.x), v * Vec(-b.v.y, a.v.y));
245     return ans;
246 }
247
248 // 直线与圆交点 depends eg, gt, V+V, V-V, V*V, d*V, len, pedal
249 vector <Point> inter(Line l, Circle C) {
250     vector <Point> ans;
251     Point P = pedal(C.O, l);
252     double h = len(P - C.O);
253     if (gt(h, C.r)) return ans;
254     if (eq(h, C.r)) {
255         ans.emplace_back(P);
256         return ans;
257     }
258     double d = sqrt(C.r * C.r - h * h);
259     Vec vec = d / len(l.v) * l.v;
260     ans.emplace_back(P + vec);
261     ans.emplace_back(P - vec);
262     return ans;
263 }
264
265 // 圆与圆的交点 depends eq, gt, V+V, V-V, d*V, len, r90c
266 vector <Point> inter(Circle C1, Circle C2) {
267     vector <Point> ans;
268     Vec v1 = C2.O - C1.O, v2 = r90c(v1);
269     double d = len(v1);
270     if (gt(d, C1.r + C2.r) || gt(abs(C1.r - C2.r), d)) return ans;
271     if (eq(d, C1.r + C2.r) || eq(d, abs(C1.r - C2.r))) {
272         ans.emplace_back(C1.O + C1.r / d * v1);
273         return ans;
274     }

```

```

275     double a = ((C1.r * C1.r - C2.r * C2.r) / d + d) / 2;
276     double h = sqrt(C1.r * C1.r - a * a);
277     Vec av = a / len(v1) * v1, hv = h / len(v2) * v2;
278     ans.emplace_back(C1.O + av + hv), ans.emplace_back(C1.O + av - hv);
279     return ans;
280 }
281
282 // 三角形的四心
283 // 三角形重心
284 Point baryCenter(Point A, Point B, Point C) {
285     return Point((A.x + B.x + C.x) / 3, (A.y + B.y + C.y) / 3);
286 }
287
288 // 三角形外心 depends r90c, V*V, d*V, V-V, V+V
289 // 给定三点求圆，要先判断是否三点共线
290 Point circumCenter(Point A, Point B, Point C) {
291     double a = A * A, b = B * B, c = C * C;
292     double d = 2 * (A.x*(B.y-C.y) + B.x*(C.y-A.y) + C.x*(A.y-B.y));
293     return 1 / d * r90c(a * (B - C) + b * (C - A) + c * (A - B));
294 }
295
296 // 三角形内心 depends len, d*V, V-V, V+V
297 Point inCenter(Point A, Point B, Point C) {
298     double a = len(B - C), b = len(A - C), c = len(A - B);
299     double d = a + b + c;
300     return 1 / d * (a * A + b * B + c * C);
301 }
302
303 // 三角形垂心 depends V*V, d*V, V-V, V^V, r90c
304 Point orthoCenter(Point A, Point B, Point C) {
305     double n = B * (A - C), m = A * (B - C);
306     double d = (B - C) ^ (A - C);
307     return 1 / d * r90c(n * (C - B) - m * (C - A));
308 }

```

### 4.6.2 极角序

直接计算极角, `atan2(y,x)`函数可直接计算 $(x,y)$ 的极角, 值域是 $(-\pi, \pi]$ 。注意第四象限的极角比第一象限要小。

```
1 db theta(Vec v) {
2     return atan2(v.y, v.x);
3 }
4
5 void psort(Points &ps, Point c = 0) {
6     sort(ps.begin(), ps.end(), [&](auto p1, auto p2) {
7         return theta(p1 - c) < theta(p2 - c);
8     });
9 }
```

先比较象限再做叉乘

```
1 int qua(Point p) { // 求象限
2     return lt(p.y, 0) << 1 | lt(p.x, 0) ^ lt(p.y, 0);
3 }
4
5 void psort(Points &ps, Point c = 0) {
6     sort(ps.begin(), ps.end(), [&](auto p1, auto p2) {
7         return qua(p1-c) < qua(p2-c) || (qua(p1-c) == qua(p2-c) && lt
8             ((p1-c) ^ (p2-c), 0));
9     });
10 }
```

这种方法常数可能稍微大一点, 但是精度比较好, 如果坐标都是整数的话是完全没有精度损失的。

### 4.6.3 凸包

#### Graham 扫描法

最左下角的一个点, 一定在凸包上, 以这个角为极点, 进行极角排序, 然后逐个点扫描。用栈来维护, 如果栈中点数小于 3, 就直接进栈; 否则, 检查栈顶三个点组成的两个向量的旋转方向是否为逆时针 (这可以用叉乘判断), 若是则进栈, 若不是则弹出栈顶, 直到栈中点数小于 3 或者满足逆时针条件为止。

实现时需要注意, 要对极角排序的极点特殊处理, 使它始终排在第一位。

```
1 // depends eq , lt , cross , V-V , P < P , len
2 using Points = vector<Point>;
3
```

```

4 bool check(Point p, Point q, Point r) { // 检查是向量旋转方向是否为逆
    时针
5     return lt(0, (q - p) ^ (r - q));
6 }
7
8 Points chull(Points &ps) {
9     sort(ps.begin(), ps.end());
10    vector<int> I{0}, used(ps.size());
11
12    for (int i = 1; i < ps.size(); i++) {
13        while (I.size() > 1 and !check(ps[I[I.size() - 2]], ps[I.back
            ()], ps[i]))
14            used[I.back()] = 0, I.pop_back();
15        used[i] = 1, I.push_back(i);
16    }
17    for (int i = ps.size() - 2, tmp = I.size(); i >= 0; i--) {
18        if (used[i]) continue;
19        while (I.size() > tmp and !check(ps[I[I.size() - 2]], ps[I.
            back()], ps[i]))
20            I.pop_back();
21        used[i] = 1, I.push_back(i);
22    }
23    Points H;
24    I.pop_back();
25    for (auto i: I)
26        H.push_back(ps[i]);
27    return H;
28 }

```

### Andrew 算法

另一种方法是不做极角排序，直接以横坐标为第一关键词、纵坐标为第二关键词排序，这样将顶点依次相连（不首尾相连）的话，也能保证不交叉。

然后同上一种方法类似，正反遍历两遍，分别求出上下凸包结合起来就好。

```

1 // depends eq , lt , cross , V-V , P < P , len
2 using Points = vector<Point>;
3
4 db theta(Point p) { // 极角
5     return p == 0 ? -1 / 0. : atan2(p.y, p.x);
6 }

```

```
7
8 void psort(Points &ps, Point c = 0) { // 极角序,先按照极角排序,再按照
    距离排序
9     sort(ps.begin(), ps.end(), [&](auto p1, auto p2) {
10         auto t1 = theta(p1 - c), t2 = theta(p2 - c);
11         if (eq(t1, t2)) return lt(len(p1 - c), len(p2 - c));
12         return lt(t1, t2);
13     });
14 }
15
16 bool check(Point p, Point q, Point r) { // 检查是向量旋转方向是否为逆
    时针
17     return lt(0, (q - p) ^ (r - q));
18 }
19
20 Points chull(Points &ps) {
21     psort(ps, *min_element(ps.begin(), ps.end()));
22     Points H{ps[0]};
23     for (int i = 1; i < ps.size(); i++) {
24         while (H.size() > 1 and !check(H[H.size() - 2], H.back(), ps[i]
            )))
25             H.pop_back();
26         H.push_back(ps[i]);
27     }
28     return H;
29 }
```

这种方法一般会比第一种快一点。



# 第五章 字符串

## 5.1 字符串哈希

### 5.1.1 单哈希

这里用unsigned long long的自然溢出做模数

```
1 typedef unsigned long long ull;
2 const int N = 1e6+5 , K = 1e9+7; // 长度 K进制
3 vector<ull> hashP(N);
4
5 void init(){// 初始化
6     hashP[0] = 1;
7     for( int i = 1 ; i < N ; i ++ ) hashP[i] = hashP[i-1] * K;
8 }
9
10 void hashStr( const string & s , vector<ull> & a){ // 计算Hash数组
11     a.resize(s.size()+1);
12     for( int i = 1 ; i <= s.size() ; i ++ )
13         a[i] = ull(a[i-1] * K + s[i-1]);
14 }
15
16 ull hashStr( const string & s ){
17     ull ans = 0;
18     for( auto i : s )
19         ans = ull( ans * K + i );
20     return ans;
21 }
22
23 ull getHash( int l , int r , const vector<ull> & a ){ //计算Hash值
24     return a[r] - a[l-1] * hashP[r-l+1];
25 }
```

### 5.1.2 双哈希

这里用unsigned long long的自然溢出和998244353做模数

```

1  #define int long long
2  #define mp make_pair
3  typedef pair<int, int> hashv;
4
5  const hashv mod = mp( 1e9+7 , 998244353 );
6  const hashv base = mp(13331,23333);
7
8  hashv operator + ( hashv a , hashv b ) {
9      int c1 = a.first + b.first , c2 = a.second + b.second;
10     if( c1 >= mod.first ) c1 -= mod.first;
11     if( c2 >= mod.second ) c2 -= mod.second;
12     return mp( c1 , c2 );
13 }
14
15 hashv operator - ( hashv a , hashv b ) {
16     int c1 = a.first-b.first , c2 =a.second-b.second;
17     if( c1 < 0 ) c1 += mod.first;
18     if( c2 < 0 ) c2 += mod.second;
19     return mp( c1 , c2 );
20 }
21
22 hashv operator * ( hashv a , hashv b ) {
23     return mp( a.first*b.first%mod.first , a.second*b.second%mod.
        second );
24 }
25
26 const int N = 2e6+5;
27
28 vector< hashv > p , hs , ht;
29
30 void hashStr( const string &s , vector<hashv> &v ){
31     v.resize(s.size()+1);
32     for( int i = 1 ; i <= s.size() ; i ++ )
33         v[i] = v[i-1] * base + mp( s[i-1] , s[i-1] );
34     return;
35 }

```

```

36 hashv getHash( int l , int r , const vector<hashv> &v){
37     if( l > r ) return mp( 0 , 0 );
38     return v[r] - v[l-1] * p[r-l+1];
39 }
40 void init( int n ){
41     p = vector<hashv>( n+1 ) , p[0] = mp(1,1);
42     for( int i = 1 ; i <= n ; i ++ ) p[i] = p[i-1] * base;
43 }

```

## 5.2 KMP

首先对字符串首先要求一个前缀函数  $\pi[i]$ 。 $\pi[i]$  简单来说就是子串  $s[0\dots i]$  最长的相等的真前缀与真后缀的长度。

```

1 vector<int> prefix_function(const string &s) {
2     int n = s.size();
3     vector<int> pi(n);
4     for (int i = 1, j; i < n; i++) {
5         j = pi[i - 1];
6         while (j > 0 && s[i] != s[j]) j = pi[j - 1];
7         if (s[i] == s[j]) j++;
8         pi[i] = j;
9     }
10    return pi;
11 }

```

然后就是 KMP 算法的实现有两种，两种做法效率实际上一样的

```

1 // pattern 在 text 中出现的位置
2 vector<int> kmp(const string &text, const string &pattern) {
3     string cur = pattern + '#' + text;
4     int n = text.size(), m = pattern.size();
5     vector<int> v, lps = prefix_function(cur);
6     for (int i = m + 1; i <= n + m; i++)
7         if (lps[i] == m) v.push_back(i - 2 * m);
8     return v;
9 }

```

除了这样做之外，还有一种做法是求不重复的匹配位置

```

1 vector<int> kmp(const string &text, const string &pattern) {

```

```

2     vector<int> v, lps = prefix_function(pattern);
3     for (int i = 0, j = 0; i < text.size(); i++) {
4         while (j && text[i] != pattern[j]) j = lps[j - 1];
5         if (text[i] == pattern[j]) j++;
6         if (j == pattern.size())
7             v.push_back(i - j + 1), j = 0;
8     }
9     return v;
10 }

```

### 5.3 Tire

```

1 struct Trie {
2     struct node {
3         vector<int> nxt;
4         bool exist;
5         char val;
6
7         node(char val = '@') : nxt(26, -1), exist(false), val(val) {};
8     };
9
10    vector <node> t;
11
12    Trie() : t(1, node()) {};
13
14    void insert(string s) {
15        int pos = 0;
16        for (auto c: s) {
17            int x = c - 'a';
18            if (t[pos].nxt[x] == -1)
19                t[pos].nxt[x] = t.size(), t.emplace_back(c);
20            pos = t[pos].nxt[x];
21        }
22        return;
23    }
24
25    bool find(string s) {
26        int pos = 0;

```

```

27         for (auto c: s) {
28             int x = c - 'a';
29             if (t[pos].nxt[x] == -1) return false;
30             pos = t[pos].nxt[x];
31         }
32         return t[pos].exist;
33     }
34 };

```

## 5.4 最小表示法

### 5.4.1 循环同构

如果字符串  $S$  选择一个位置  $i$  满足

$$S[i...n] + S[1...i-1] = T$$

则称  $S$  与  $T$  循环同构

### 5.4.2 最小表示法

对于一对字符串  $A, B$ ，他们在原串中的起始位置分别为  $i, j$ ，且前  $k$  个字符均相同，即

$$S[i...i+k-1] = S[j...j+k-1]$$

若  $S[i+k] > S[j+k]$ ，则其实下表  $l \in [i, i+k]$  的字符串均不可能为最优解，因为如果有  $l = i+p$  则一定有  $j+p$  字典序更小，所以可以直接把  $i$  移动到  $i+k+1$  进行比较。

```

1  int minNotation(const vector<int> &s) {
2      int n = s.size();
3      int i = 0, j = 1;
4      for (int k = 0; k < n && i < n && j < n;) {
5          if (s[(i + k) % n] == s[(j + k) % n]) k++;
6          else {
7              if (s[(i + k) % n] > s[(j + k) % n]) i = i + k + 1;
8              else j = j + k + 1;
9              if (i == j) i++;
10             k = 0;
11         }
12     }
13     return min(i, j);
14 }

```



# 第六章 动态规划

## 6.1 线性 DP

## 6.2 背包

### 6.2.1 01 背包输出方案

```
1 // dp 时
2 // c[i] 是价格 , w[i] 是价值
3 for(int i = 0 ; i < N ; i++) {
4     for(int j = V ; j >= c[i] ; j++)
5         if(f[j-c[i]]+w[i] > f[j]) {
6             f[j] = f[j-c[i]]+w[i];
7             path[i][j] = 1;
8         }
9 }
10
11 // 输出方案
12
13 int i = N, j = V;
14 while(i > 0 && j > 0) {
15     if(Path[i][j] == 1) {
16         cout << c[i-1] << " ";
17         j -= c[i-1];
18     }
19     i--;
20 }
```

## 6.3 树形 DP

### 6.3.1 普通树形 DP

给一颗  $n$  个节点有根的树，树上标号  $i$  的点权值为  $h_i$ 。在树上选一些点，要求父节点和子节点不能同时选。问权值和最大是多少？

$f[i][0]$  表示在  $i$  子树中选择，不选  $i$  的最大权值和， $f[i][1]$  表示选  $i$  的最大权值和。

对于一对父子  $(x, y)$ ，如果选父节点  $x$ ，则  $y$  不能选， $f[x][1] += f[y][0]$ 。如果不选  $x$ ，则  $y$  随意， $f[x][0] += \max(f[y][1], f[y][0])$ 。注意选  $x$  还要加上本身， $f[x][1] += h[x]$ 。

```

1 // NC1044A
2 #include <bits/stdc++.h>
3 using namespace std;
4 int read() { ... }
5
6 int n, root;
7 vector<bool> v;
8 vector<int> h;
9 vector <vector<int>> e, f;
10
11 void dp(int x) {
12     f[x][1] = h[x];
13     for (auto y: e[x]) {
14         dp(y);
15         f[x][1] += f[y][0];
16         f[x][0] += max(f[y][0], f[y][1]);
17     }
18 }
19 int32_t main() {
20     n = read();
21     v = vector<bool>(n + 1, 0), h = vector<int>(n + 1);
22     e = vector < vector < int >> (n + 1);
23     f = vector < vector < int >> (n + 1, vector<int>(2));
24     for (int i = 1; i <= n; i++) h[i] = read();
25     for (int i = 1, x, y; i < n; i++)
26         x = read(), y = read(), e[y].push_back(x), v[x] = 1;
27     for (int i = 1; i <= n; i++) {
28         if (v[i]) continue;

```



```

29         root = i;
30         break;
31     }
32     dp(root);
33     cout << max(f[root][0], f[root][1]);
34     return 0;
35 }

```

### 6.3.2 背包类树形 DP

给一颗大小为  $n$  树，树上每个点都有一个点权。选择节点的先决条件是选择其父节点。最多可以选择  $m$  个节点，问可选的最大权值是多少。

设  $f[i][j]$  表示  $i$  节点子树中选择不超过  $j$  个节点的最大权值。从儿子转移过来的过程其实就是一个类似**分组背包**的过程。

```

1 // NC1044B
2 void dp(int x) {
3     f[x][0] = 0;
4     for (auto y: e[x]) {
5         dp(y);
6         for (int i = m; i >= 0; i--) // 枚举当前节点最大可用背包容积
7             for (int j = i; j >= 0; j--) // 枚举当子节点最大可用背包容积
8                 f[x][i] = max(f[x][i], f[x][i - j] + f[y][j]);
9     }
10    // 这里要给每种容积都加上他本身的权值。
11    for (int i = m; x != 0 && i > 0; i--) f[x][i] = f[x][i-1] + val[x];
12    return;
13 }

```

### 6.3.3 换根 DP

给定一个  $n$  个点的树，请求出一个结点，使得以这个结点为根时，所有结点的深度之和最大。

```

1 // luogu P3478
2 #include<bits/stdc++.h>

```

```
3 using namespace std;
4 #define int long long
5
6 int read() {...}
7
8 int n;
9 vector<int> d, f, son;
10 vector<bool> vis;
11 vector<vector<int>> e;
12
13 void dfs(int x) {
14     vis[x] = 1;
15     for (auto y: e[x]) {
16         if (vis[y]) continue;
17         d[y] = d[x] + 1;
18         dfs(y);
19         son[x] += son[y];
20     }
21     return;
22 }
23
24 void dp(int x){
25     vis[x] = 1;
26     for( auto y : e[x] ){
27         if( vis[y] ) continue;
28         f[y] = f[x] + n - 2*son[y];
29         dp(y);
30     }
31 }
32
33
34 int32_t main() {
35     n = read();
36     d = vector<int>(n + 1), f = vector<int>(n + 1);
37     son = vector<int>(n + 1, 1), son[0] = 0;
38     vis = vector<bool>(n + 1);
39     e = vector<vector<int>>(n+1);
40     for (int i = 1, u, v; i < n; i++)
```

```

41     u = read(), v = read(), e[u].push_back(v), e[v].push_back(u);
42     dfs(1);
43     vis = vector<bool>(n + 1);
44     for( int i = 1 ; i <= n ; i ++ )
45         f[1] += d[i];
46     dp(1);
47     int val = 0 , res = 0;
48     for( int i = 1 ; i <= n ; i ++ )
49         if( f[i] > val ) val = f[i] , res = i;
50     cout << res;
51     return 0;
52 }

```

### 6.3.4 最大独立集

一条边的两个端点只能选一个，问最多选多少个点。

```

1 // NC51178
2 int32_t main() {
3     int n;
4     cin >> n;
5     vector<int> h(n + 1);
6     for (int i = 1; i <= n; i++) cin >> h[i];
7     vector<vector<int>> e(n + 1);
8     int root = n * (n + 1) / 2;
9     for (int i = 1, x, y; i < n; i++)
10         cin >> y >> x, e[x].push_back(y), root -= y;
11     vector<array<int, 2>> f(n+1);
12     auto dfs = [e, h, &f](auto &&self, int x) -> void {
13         f[x][1] = h[x];
14         for (auto y: e[x]) {
15             self(self, y);
16             f[x][0] += max(f[y][0], f[y][1]);
17             f[x][1] += f[y][0];
18         }
19         return;
20     };
21     dfs(dfs, root);
22     cout << max(f[root][1], f[root][0]);

```

```
23     return 0;
24 }
```

### 6.3.5 最小点覆盖

选一个点可以把相邻的边覆盖，问最少选多少个点可以把所有的边覆盖。

```
1 // NC 51222
2 int32_t main() {
3     int n;
4     while (cin >> n) {
5         vector<vector<int>> e(n);
6         vector<array<int, 2>> f(n);
7         int root = n * (n - 1) / 2;
8         for (int x, y, t, i = 1; i <= n; i++)
9             for( x = read() , t = read() ; t ; t -- )
10                 y = read(), e[x].push_back(y), root -= y;
11         auto dfs = [e, &f](auto &&self, int x) -> void {
12             f[x][1] = 1;
13             for (auto y: e[x]) {
14                 self(self, y);
15                 f[x][1] += min(f[y][0], f[y][1]);
16                 f[x][0] += f[y][1];
17             }
18         };
19         dfs(dfs, root);
20         cout << min(f[root][1], f[root][0]) << "\n";
21     }
22     return 0;
23 }
```

### 6.3.6 最小支配集

选一个点可以把相邻的点覆盖，问最少选多少个点可以把所有的点覆盖。

```
1 // NC 24953
2 int32_t main() {
3     ios::sync_with_stdio(0), cin.tie(0);
4     int n;
5     cin >> n;
```

```

6      vector<vector<int>>> e(n + 1);
7      for (int i = 1, x, y; i < n; i++)
8          cin >> x >> y, e[x].push_back(y), e[y].push_back(x);
9      vector<array<int, 3>> f(n + 1);
10     // f[x][0] x 被自己覆盖, f[x][1] x 被儿子覆盖, f[x][2] x 被父亲覆盖
11     auto dfs = [&f, e](auto &&self, int x, int fa) -> void {
12         f[x][0] = 1;
13         f[x][1] = inf;
14         f[x][2] = 0;
15         int inc = inf;
16         for (auto y: e[x]) {
17             if (fa == y) continue;
18             if (f[x][1] == inf) f[x][1] = 0;
19             self(self, y, x);
20             f[x][0] += min({f[y][0], f[y][1], f[y][2]});
21             f[x][2] += min(f[y][0], f[y][1]);
22             f[x][1] += min(f[y][0], f[y][1]), inc = min(inc, f[y][0] -
                    f[y][1]);
23         }
24         f[x][1] += max(0, inc);
25         return;
26     };
27
28     dfs(dfs, 1, -1);
29     cout << min(f[1][0], f[1][1]);
30     return 0;
31 }

```

### 6.3.7 求任意子树的直径

```

1  int32_t main() {
2      int n;
3      cin >> n;
4      vector<int> val(n + 1);
5      vector<vector<int>>> e(n + 1);
6      for (int i = 1; i <= n; i++) cin >> val[i];
7      for (int i = 1, x, y; i < n; i++)
8          cin >> x >> y, e[x].push_back(y), e[y].push_back(x);

```

```

9      vector<int> f(n + 1 , - inf ), dis(n + 1);
10     // f[x] 表示 x 的子树的直径 , dis[x] 表示 x 向下最远可以走多远
11     auto dfs = [e, val, &f, &dis](auto &&self, int x, int fa) -> void
        {
12         multiset<int> t;
13         for (auto y: e[x]) {
14             if (y == fa) continue;
15             self(self, y, x);
16             f[x] = max(f[x], f[y]);
17             dis[x] = max(dis[x], dis[y] + val[y]);
18             t.insert(dis[y] + val[y]);
19             if (t.size() == 3) t.erase(t.begin());
20         }
21         int w = val[x];
22         for (auto i: t) w += i;
23         f[x] = max(f[x], w);
24         return;
25     };
26     dfs(dfs, 1, -1);
27     return 0;
28 }

```

## 6.4 状态压缩 DP

### 6.4.1 TSP 问题

给一个有权图，求一条代价和最小的回路，使得该回路恰好经过每个点一次

复杂度  $O(n^2 2^n)$

```

1 // https://www.acwing.com/problem/content/93/
2 // 这道题给了额外的限定，要求起点必须从 0 开始
3 #include <bits/stdc++.h>
4 using namespace std;
5 #define int long long
6 const int N = 20, M = 1 << N, inf = 1e17;
7 int e[N][N], f[M][N], n;
8
9 int calc(int st, int i) {
10     if (f[st][i] != inf) return f[st][i];

```

```
11     f[st][i] --; // 标记当前状态被访问过了，以免当前状态无解被反复访问
12     int p = st - (1 << i);
13     for (int j = 0; j < n; j++) {
14         if ((p & (1 << j)) == 0) continue;
15         f[st][i] = min(f[st][i], calc(p, j) + e[j][i]);
16     }
17     return f[st][i];
18 }
19
20 int32_t main() {
21     ios::sync_with_stdio(0), cin.tie(0);
22     cin >> n;
23     for (int i = 0; i < n; i++)
24         for (int j = 0; j < n; j++)
25             cin >> e[i][j]; // 边权
26     fill(f[0], f[0] + M * N, inf) , f[1][0] = 0;
27     cout << calc( ( 1 << n ) - 1 , n - 1 );
28     return 0;
29 }
```